KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering

Design and Development of Probabilistic Inference Pipelines

Dimitar Shterionov

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

August 2015

Design and Development of Probabilistic Inference Pipelines

Dimitar SHTERIONOV

Examination committee:
Prof. dr. ir. Joseph Vandewalle, chair
Prof. dr. ir. Gerda Janssens, supervisor
Prof. dr. ir. Maurice Bruynooghe
Prof. dr. ir. Marie-Francine Moens
Prof. dr. ir. Hendrik Blockeel
Prof. dr. Ricardo Rocha
(University of Porto, Portugal)
Dr. ir. Jan Wielemaker
(University of Amsterdam, the Netherlands)
Prof. dr. Bart Demoen
(KU Leuven, Belgium)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

August 2015

KU Leuven – Faculty of Engineering, Department of Computer Science,
Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

ISBN 978-954-745-253-4

To my parents.

Preface

"If a coin comes down heads, that means that the possibility of its coming down tails has collapsed. Until that moment the two possibilities were equal.

But on another world, it does come down tails. And when that happens, the two worlds split apart."

Philip Pullman

I started my PhD approximately four and a half years before the last page of this manuscript. Four and a half years that passed like four and a half days. The road I walked during my PhD was curvy and I faced a lot of obstacles but it also brought me joyful experiences, introduced me to interesting people and gave me great opportunities. During these four and a half years I worked on the development of the probabilistic software ProbLog – a rapidly evolving system that could eventually answer the hardest questions ... probabilistically.

As a PhD student I learned from textbooks and papers but the most I learned from the meetings and discussions with the people of the Computer Science department, KU Leuven. And most of all, with my promoter professor Gerda Janssens. I would like to express my deepest gratitude to her for the guidance in solving both theoretical and practical questions, her support when facing obstacles, her honesty and her patience. I admire her ability to analyze an idea from every aspect and point out the slightest details that would lead to the correct realization of that idea, if that is possible.

I would also like to express my sincere gratitude to professor Maurice Bruynooghe who helped me with different matters throughout my PhD and especially with finalizing this manuscript.

I would like to thank the members of my examination committee: professor Bruynooghe, professor Moens, professor Blockeel, professor Rocha, dr.

Wielemaker and professor Demoen for devoting precious time to read this manuscript and for providing constructive comments for improving my work. Furthermore, I would like to thank the chairman of this committee professor Joseph Vandewalle.

ProbLog is not a one-man project. It is the result of the collaboration of many people. I would like to thank my colleagues from the ProbLog team – Jonas, Joris, Dries, Anton, Wannes, Guy and Angelika. Special thanks goes to Angelika – the most just person I have ever met. Also, I would like to thank Theofrastos who was my daily supervisor during my master studies and a colleague and a friend during my PhD studies. With him I worked on different topics often staying late in the nigh drawing ideas on the white board in the office or on a napkin in a restaurant. Also I am thankful to Jan Van Haaren, and Daan Fierens with whom I shared an office for the first 2 years of my studies.

My time at KU Leuven would not been as nice without the people from the $13h30$ lunch group and the coffee-break group. Thank you, Gitte, Nick, Nima, Koosha, Theo, Tobias, Tom, Tomas, Zubair, Jerome, Christos, Ondrej and Lilian for the pleasant time full with interesting discussions. Special thanks goes to Gitte for the pleasant coffee breaks, ping pong and pool games and for being there to talk about PhD related and unrelated matters.

Except for the professional environment at KU Leuven, I spend the last 4.5 years in the company of a lot of friends from places all over the world. I would like to say thank you to Deni, Ivo and Stefan who are not only volleyball players, barbecue organizers and motor bikers but also musketeers; to Hrissi for the chocolate, the ice cream ... and the rain; to Yana, too; to the guitar players with whom I annoyed on regular basis the neighbors often until sunrise – Julio and Petar; and also to Jochen, Velina, Ani, Vesko, Stoyan, Daro, Dimiter and the rest of my friends here or far away.

I would also like to express my deep gratitude to my family – Shteliyan, Elinka and Aleksandar, who stood by me even being far away. Thank you mom, dad, brother for the support and the strength that you gave me.

Last, but not the least I would like to say thank you to my girlfriend Eva, who brought a great amount of love and happiness into my life. She helped me through the writing of this manuscript and supported me in difficult moments; she listened to my ProbLog and Prolog, and Python, and Latex talks that probably sounded completely boring to her, and still she was right by my side.

At the end I would like to thank all of you who read this textbook.

Abstract

Artificial Intelligence (AI) is a broad scientific field which aims at the study and the development of computer systems that can simulate human behavior. Naturally, logic is the fundament of many such systems as it provides an intuitive mechanism to represent knowledge. Logic can be used to address a wide range of AI applications. But for applications that require reasoning with uncertain knowledge more profound formalisms are needed. This necessity lead to the establishment and the vast development of the fields of Probabilistic Logic Programming (PLP), Statistical Relational Learning (SRL) and others.

ProbLog is a PLP framework – a language and an inference system that started as a probabilistic extension of Prolog. Soon after, it established a dominant position within the PLP community. In this thesis we present recent advances in the design and the development of ProbLog. We focus on the ProbLog2 system which aims to bridge the gap between the PLP and SRL communities by implementing typical SRL tasks while using the general ProbLog language. In general, probabilistic inference is a computationally expensive task. In order to solve realistic problems efficiently, the ProbLog system implements a pipeline architecture, called the ProbLog inference pipeline. It applies a sequence of transformation steps encapsulated in four separate components, in order to reduce the inference task to a simple weighted model counting (WMC) problem by means of knowledge compilation. Each component can be implemented with different tools or algorithms. The modularity of this architecture allows to (i) substitute the implementation of one component with another; (ii) extend the inference pipeline with new components or processing steps; and (iii) build new inference and learning tasks with minimum efforts.

In this thesis we first introduce the ProbLog inference pipeline, discuss existing implementations, present new ones and evaluate their performance. As a consequence we determine crucial points for its efficiency. Then we focus on its optimization. We present a method that aims at improving knowledge compilation by compacting the input Boolean formulae. Our method detects

seven subformulae patterns and uses them to rewrite a given Boolean formula into a more compact representation. These patterns are associated with two types of logic transformations – one type that retains logic equivalence and another type that retains the WMC. Although our approach was implemented in the scope of ProbLog it is more general and any application that uses Boolean formulae to represent knowledge can benefit from it. Next, we augment the inference pipeline of ProbLog2 to handle two extensions of the ProbLog language – constraints and annotated disjunctions. Both constraints and annotated disjunctions increased the expressive power of the ProbLog language: constraints are First-Order Logic sentences which need to hold; annotated disjunctions provide an intuitive way to encode random events with multiple and mutually exclusive outcomes. To incorporate constraints in ProbLog we devised a method that (i) converts constraints into ProbLog syntax and (ii) uses the default ProbLog2 pipeline to perform inference. Annotated disjunctions had already been incorporated in the ProbLog language by means of an encoding that was correct for some of the inference tasks that ProbLog supports but incorrect for more recent ones. In order to provide support for annotated disjunctions that is correct for all inference tasks of ProbLog2 we devised a constraint-based method to encode annotated disjunctions as ProbLog programs. The modularity of the inference pipeline enabled a systematic design and implementation of these approaches.

Abbreviations

AC	Arithmetic Circuit, 34
AD	Annotated disjunction, 136
BDD; ROBDD	Binary Decision Diagram; Reduced Ordered Binary Decision Diagarms, 32
BN	A Bayesian network, 126
CHRiSM	CHance Rules induce Statistical Models, 130
CNF	Conjunctive Normal Form, 21
COND	Conditional probability task, 13
CP	Causal-Probabilistic, 137
CPT	Conditional probability table, 127
CWA	Closed World Assumption, 28
DNF	Disjunctive Normal Form, 24
FOL	First-Order Logic, 28
KC	Knowledge Compilation, 19
LFI	Learning from interpretations task, 14
LHM	Least Herbrand Model, 36
LP	Logic Programs, 28
LPADs	Logic Programs with Annotated Disjunctions, 137
MAP	Maximum a posteriori task, 14
MARG	Marginal probability task, 13
MPE	Most probable explanation task, 14

NNF	Negation Normal Form, 20
OLDT resolution	Ordered selection strategy with Linear resolution for Definite clauses with Tabling, 25
P0, .., P13	ProbLog inference pipelines, 36
PCLP	Probabilistic Constraint Logic Programming, 123
PGM	Probabilistic graphical models, 141
PLP	Probabilistic Logic Programming, 10
PNF	Prenex Normal Form, 104
SAT	Boolean Satisfiability Problem, 19
sd-DNNF	Smooth Deterministic Decomposable Negation Normal Form, 32
SLD resolution	Selective Linear Definite clause resolution, 24
SLDNF resolution	Selective Linear Definite clause with Negation as Failure, 24
SLG resolution	Linear resolution with Selection function for General logic programs, 25
SRL	Statistical Relational Learning, 19
VIT	Most probable proof task, also called Viterbi proof, 143
wCNF	Weighted Boolean formula in CNF, 145
WMC	Weighted Model Counting, 19

List of Symbols

$(p_t, p_f) :: \texttt{spf}(\texttt{a}_\texttt{j}, \texttt{h}_\texttt{i}, \texttt{i})$	Surrogate probabilistic fact with true probability p_t and false probability p_f
λ_{a_i}	Indicator variable of the atom a_i
$\wedge$	Conjunction, AND
$\Leftrightarrow$	Equivalence
$\neg$	Negation, NOT
$\vee$	Disjunction, OR
$\mathcal{F}_L$	Program function
Ω	Set of possible worlds
ω_i; ω^q	Possible world; possible world that entails query q
$\mathcal{S}$	CHRiSM constraint store
$\mathcal{T}$; $\mathcal{T}^{PCLP}$; $\mathcal{T}^{FOProbLog}$	A Theory; a PCLP theory; a FOProbLog theory
$\Rightarrow$	Implication
σ_i; $\mathcal{S}$	Selection; set of selections
θ_{a_i}	Weight variable of atom a_i
φ; ψ	Boolean formula
$\veebar$	Exclusive OR
At	Set of ground atoms
c; C	Constraint; set of constraints

D_g	Dependency set of goal g
$E = e$	Evidence, where E is a set of atoms and e a vector of truth values
g, g_i	Goal
G; V; E	Graph; set of vertexes (or nodes); set of edges
I	Interpretation
L	ProbLog program
L^C	cProbLog program
$L^{CLP(BN)}$	CLP($\mathcal{BN}$) program.
$P(\omega_i)$	Probability of possible world ω_i according to the ProbLog semantics
$P(q)$	Marginal probability of query q
$P_C(\omega_i)$	Probability of possible world ω_i according to the cProbLog semantics
$p_i :: f_i$	Probabilistic fact f_i with probability p_i
$P_{min}(q)$; $P_{max}(q)$	Minimum and maximum probability of a query atom q
q; Q	Query atom; set of query atoms
$R1$; $R2$; $R3$	Rewrite rules
sg, sg_i	Subgoal
T_x	Execution time for process x
$Tr(A)$	Probability tree of the set of annotated disjunctions A
$val(\mathcal{F}, \alpha)$	Value of program function $\mathcal{F}$ given the value α to all uninstantiated variables
BK	Background Knowledge
F; $false$	False
prior, post, both	Compaction settings
T; $true$	True

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Logic Programming (LP) is a programming paradigm apt to model and reason about problems over relational data. Logic programs consist of statements that define what is true for a given domain. Although a wide range of AI applications use LP solutions, the necessity to efficiently reason with uncertain knowledge has lead to the foundation of more profound formalisms such as Probabilistic Logic Programming (PLP). PLP encompasses an ever-growing set of techniques and software tools that extend logic programming with probabilistic reasoning. Probabilistic logic programs extend logic programs and consider statements that are not certain to be true (or false) but are assigned a probability. PLP languages include ProbLog [13], PRISM [68], LPADs [87], PHA [58] and others. Most of them build upon an existing LP language, such as Prolog. They inherit syntax and semantics from the original language; the syntax is augmented with constructs to encode probabilistic knowledge and the semantics is extended to handle these constructs. The language ProbLog extends Prolog with probabilistic facts. Probabilistic facts encode independent random variables that can be assigned one of two values, i.e., true or false. Probabilistic facts can express stochastic events with two outcomes. LPADs, which stands for Logic Programs with Annotated Disjunctions use annotated disjunctions to encode multi-valued random variables. Annotated disjunctions can be used to express stochastic events with multiple and mutually exclusive outcomes. Annotated disjunctions are similar to the multi value switches employed in the PRISM (short from PRogramming In Statistical Models) language and the disjoint sets employed in the PHA (Probabilistic Horn Abduction) language. From semantical perspective, PRISM and PHA impose certain restrictions on their programs. For example, all rules defining a predicate must be mutually exclusive, and programs must be acyclic. The ProbLog language stands out

among the rest as (i) it is less restrictive – a ProbLog program can be cyclic, contain negation (over probabilistic facts as well as general negation) and no rules should be mutually exclusive; and (ii) extensions of the ProbLog language increase its expressivity – it supports annotated disjunctions [21] and also meta predicates [45]. The semantics of ProbLog is based on Sato's distribution semantics [65] which defines a joint probability distribution over the possible least Herbrand models.

The main inference task for PLP is to determine the probability that a query is true given a probabilistic logic program – referred in the PLP literature as the *success* probability.

In parallel to PLP, the field of Statistical Relational Learning (SRL) [20] has evolved to tackle the same problems from a different perspective. While PLP extends logic programming with probabilistic reasoning and focuses on the success probability of a query, the field of SRL incorporates logical and relational representations into graphical models, e.g., probabilistic relational models [17], Bayesian logic programs [29], and others. The main inference tasks in SRL are computing the *marginal* probability of random variables, the *conditional* probability of a set of random variables given evidence and finding the *most probable explanation* given evidence and the *maximum a-posteriori* probability. A common learning task is to learn the maximum likelihood parameters from (possibly partial) interpretations.

We often refer to ProbLog as a framework that consists of the ProbLog language and the ProbLog inference and learning system. The ProbLog system was first implemented [13, 34] as a typical PLP tool focusing on the task to compute the *success* probability of a query. We refer to this system as ProbLog1. The next generation of the system, ProbLog2 [14, 16], focuses on a wider range of inference and learning tasks. ProbLog2 aims to bridge the gap between PLP and SRL by implementing the inference tasks of computing the conditional probability of a query given evidence and the most probable explanation; and the learning from interpretations task; the success probability task, common in the PLP community, coincides with the task to compute the marginal probability, typical for the SRL community. The ProbLog language, initially built to support only probabilistic facts, now extends over annotated disjunctions, but also to constraints.

Probabilistic inference and learning are computationally expensive tasks. To perform them efficiently the ProbLog system employs knowledge compilation to reduce the inference or learning task to a computationally inexpensive weighted model counting problem. The system implements a pipeline architecture called an *inference pipeline* that performs a sequence of transformation steps. These transformations are encapsulated in four pipeline *components*. Each

of these components can be implemented by different tools or algorithms once the input/output requirements are respected. For example, ProbLog1 uses knowledge compilation to Reduced Ordered Binary Decision Diagrams (ROBDDs) [5]; ProbLog2 uses knowledge compilation to Smooth Deterministic Decomposable Negation Normal Form (sd-DNNF) [12]. The efficiency of a ProbLog inference pipeline depends not only on the implementation of each separate component but also on how the data exchanged by these components is represented and the task to be solved. For instance, the inference pipeline of ProbLog2 handles the learning from interpretations task better than ProbLog1 given the properties of sd-DNNFs compared to ROBDDs.

The modularity of the pipeline architecture of ProbLog allows on the one hand to easily substitute the implementation of one component with another and on the other hand to extend with other components or processing steps in order to improve the performance and the usability of the software.

Contributions

This thesis focuses on the design and development of the ProbLog framework. We researched techniques to improve the performance of the ProbLog inference pipeline as well as to support new inference tasks and language constructs. In particular the contributions of this thesis are:

- The analysis of the ProbLog inference pipeline with main focus on the development of the default ProbLog2 pipeline. Moreover, we combine the different implementations of ProbLog to achieve new pipelines. In particular, we present 14 different inference pipelines. Then we evaluate their performance to determine which one is optimal and under what conditions. One of these pipelines, that combines implementations of components of ProbLog2 and of MetaProbLog [45], outperforms the rest on many benchmarks. We build upon the work in [71, 72].
- A variable compaction method for ProbLog programs that aims to improve knowledge compilation. Since knowledge compilation is the most computationally expensive step optimizing its performance will, naturally, have a positive impact on the inference. The method we propose detects and compacts subformula patterns in Boolean formulae. Compaction is used during probabilistic inference in order to reduce the size of Boolean formulae that are used for knowledge compilation. Consequently this reduction has an effect on the knowledge compilation and evaluation run times. Although we implemented our approach as part of the ProbLog inference pipeline and used typical ProbLog problems to evaluate its performance, it is more general

and any software that uses Boolean formulae to represent knowledge can benefit from it. We build upon the work presented in [74, 47].

- The idea of extending ProbLog with constraints and the corresponding semantics were introduced in [15]; they provide neither the syntax nor an implementation of this extension. In our work we designed and developed the first inference approach for this extension. It defines a set of rewriting rules to convert a cProbLog program, that is a ProbLog program with constraints, into a ProbLog program with evidence that is semantically equivalent. Our approach reduces the computation of the probability of a query (or a set of queries) given that the constraints hold into computing the conditional probability of the query (or queries) given evidence. Although we implemented cProbLog on top of the existing ProbLog2 inference pipeline, our method to reason with cProbLog constraints can easily by incorporated in other systems. We illustrate this by adding cProbLog constraints to $CLP(\mathcal{BN})$ [64]. We also present an optimization method for cProbLog inspired by our Boolean formula compaction approach. We build upon the work in [70]

- The ProbLog language supports annotated disjunctions from its earliest implementations [21]. Annotated disjunctions impose some requirements on the inference system. In particular, while the default encoding of annotated disjunctions was correct for the success or marginal probability task and the conditional probability task, it was not correct for the most probable explanation task. We devised a new encoding based on cProbLog constraints that is correct for all inference or learning tasks currently supported by ProbLog. Then we added support for the most probable explanation task. Our implementation is based on knowledge compilation. We build upon the work presented in [75].

The rest of the text is organized in two parts. **Part I** is about the design and the implementation of the ProbLog inference pipeline. In Chapter 2 we discuss the ProbLog language – syntax and semantics, and analyze the pipeline architecture of a ProbLog system. We present and study 14 inference pipelines. Then we summarize the results of extensive experiments that we performed in order to determine which pipeline is optimal and under what conditions. Chapter 3 presents our method to compact Boolean formulae. We discuss our implementation and the experimental results. **Part II** focuses on the extension of the ProbLog language with constraints and annotated disjunctions and the relevant support for inference with these new language constructs. Chapter 4 is devoted on the extension of the ProbLog language with constraints. It introduces the first implementation of cProbLog, that is, ProbLog with constraints, and a series of cProbLog programs used to exemplify this extension.

In Chapter 5 we illustrate how we use constraints to devise a method to encode annotated disjunctions as ProbLog programs. The approach we propose retains the semantics of annotated disjunctions when encoded as ProbLog programs. This semantic equivalence enables the ProbLog system to perform all inference and learning tasks on ProbLog programs with annotated disjunctions, including the most probable explanation task.

Part I

The ProbLog System

Outline Part I

In this part we present the ProbLog programming language – its syntax and semantics – and focus on the mechanisms to perform probabilistic inference. In order to do inference efficiently, ProbLog systems implement a pipeline architecture the we call *inference pipeline*. We identify four main components – *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. Each of these components performs a certain transformation (or a set of transformations) so that the expensive inference task is reduced to an efficient weighted model counting problem. The starting point of an inference pipeline is a ProbLog program together with a (possibly empty) set of queries and a (possibly empty) set of evidence atoms, and an inference task to be solved.

In **Chapter 2** we introduce the notion of a ProbLog inference pipeline, and present more than 45 possible implementations. Then we give an extensive analysis of each pipeline component and determine 14 inference pipelines that can perform efficiently on real-world applications. Among these 14, 5 pipelines are new, i.e., have not been used for ProbLog inference before. We conduct extensive experiments on these 14 pipelines and determine on the one hand crucial components in a pipeline and on the other hand, the reasons and conditions for a particular pipeline to be preferable than the others. One of the newly introduced pipelines shows to be very promising on our benchmarks.

Then, in **Chapter 3**, we present a method to optimize the intermediate results that are conveyed between the adjacent components in an inference pipeline. This method aims to improve the performance of the knowledge compilation component by reducing the size of the input Boolean formula. Our method is a two-step procedure to first detect regularities in the Boolean formula and second to transform these regularities into more compact representations. We implemented our *detection/compaction* method in 6 inference pipelines. We performed extensive empirical evaluation of our approach to determine how effective is our method in practice.

Chapter 2

ProbLog Inference Pipeline

In order to handle real-world problems, state-of-the-art probabilistic logic and learning frameworks reduce the expensive inference task to an efficient Weighted Model Counting. To do so, ProbLog [13, 32, 14] employs a sequence of transformation steps, called an *inference pipeline*. Each step in the probabilistic inference pipeline is called a *pipeline component*. We identify four main components in a ProbLog inference pipeline – *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. Given a ProbLog program and an inference task to solve each component applies a certain transformation and produces input for the next component. The choice of the mechanism to implement a component can be crucial to the performance of the system – either due to the implementation of the component or due to the compatibility of the output with the input requirements of the next component.

To investigate how the implementation of each component affects the overall performance of the inference pipeline, first we perform a systematic analysis of the existing tools and mechanisms employed by a component. Such a thorough analysis was never performed for ProbLog inference pipelines. Second, we select 14 inference pipelines – 9 are already used by mainstream ProbLog implementations such as ProbLog1 [13], MetaProbLog [45] and ProbLog2 [14]; we introduce 5 new pipelines. Then we test these pipelines on a wide range of benchmarks. These benchmarks are standard ProbLog applications that have been previously used to evaluate ProbLog implementations. One of the newly introduced pipelines shows very promising results on our benchmarks.

Our analysis and experimental results allow us to determine which components have a crucial impact on the overall system performance. We identify that the Boolean formula conversion is a crucial component in a probabilistic

inference pipeline. In particular, the output Boolean formula strongly affects the Knowledge compilation, which is computationally the most expensive step in the pipeline.

This chapter is based on the work presented in [71, 72]. It is structured as follows: first we present the syntax and semantics of ProbLog in Section 2.1; next, we analyze the different inference pipelines in Section 2.2; and finally we present our experimental results in Section 2.3 and conclude in Section 2.4.

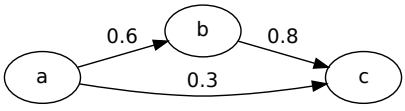
2.1 The Probabilistic Logic Programming Language ProbLog

2.1.1 Syntax

ProbLog [13, 32] is a general purpose Probabilistic Logic Programming (PLP) language. It extends Prolog with probabilistic facts which encode uncertain knowledge. Probabilistic facts can be ground or non-ground. Similar to [32], we restrict our discussion to programs with finitely many ground probabilistic facts. Probabilistic facts are the most basic constructs for encoding uncertain data. They have the form $p_i :: f_i$ and state that the fact f_i is true with probability p_i or false with probability $(1 - p_i)$. Prolog rules define logical consequences of the probabilistic facts.

Example 2.1 shows a ProbLog program that encodes a probabilistic graph – a graph whose edges are labeled with a probability. A wide range of data mining and machine learning problems can be seen as probabilistic graphs and thus encoded as ProbLog programs. ProbLog systems[1] have been used in applications like system prognostics and diagnostics [88], link and node prediction in biological data [31], robotics [52] and others.

Example 2.1 (3-node acyclic graph)**.** *A probabilistic graph models the uncertain dependencies between two entities. The entities are represented as nodes and each dependency as an edge labeled with the probability that the dependency holds.*



[1] When it is clear from the context, we use the term ProbLog to refer to either the language or the system. Otherwise we state it explicitly.

A ProbLog program can encode such a graph.

```
0.6::e(a, b).   0.3::e(a, c).  0.8::e(b, c).
p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

The fact `0.6::e(a, b).` *expresses that the edge from node* `a` *to node* `b` *exists with probability* 0.6 *(and with probability* $1.0 - 0.6 = 0.4$ *it can be missing). All possibly existing edges in the graph are expressed by the corresponding probabilistic facts; non existing edges, e.g., from node* `b` *to node* `a` *are omitted in the ProbLog program or can be expressed as a probabilistic fact with probability* 0.0*, e.g.,* 0.0`::e(b, a).`*. The* `p/2` *predicate defines the ("path") relation between two nodes: a path exists, if two nodes are connected by an edge or via a path to an intermediate node.* △

An atom that unifies with a (ground) probabilistic fact is called a (ground) *probabilistic atom*. An atom that unifies with the head of a (ground) rule is called a (ground) *derived atom*. The sets of probabilistic and derived atoms of a ProbLog program should be disjoint.

Example 2.2. *The set of all ground probabilistic atoms for the ProbLog program encoding the 3-node acyclic graph (Example 2.1) is:*

$$\{\texttt{e(a, b)}, \texttt{e(a, c)}, \texttt{e(b, c)}\}.$$

And the set of ground derived atoms entailed by the program is:

$$\{\texttt{p(a, b)}, \texttt{p(a, c)}, \texttt{p(b, c)}, \texttt{p(a, c)}\}.$$ △

In addition to probabilistic facts the language used by the ProbLog2 system supports *intensional probabilistic facts*. Intensional probabilistic facts are used to compactly define a set of probabilistic facts with the same functor and arity. They have the form $P :: f(X_1, .., X_n)$:- *Body*., where *Body* is a conjunction of literals and does not include other probabilistic facts. P is the probability label. It can be either a number or a variable which unifies with a number when the body is proven and is called a *flexible probability*. The variables X_1 to X_n are instantiated when the body is proven.

Example 2.3. *Consider the intensional probabilistic fact:*
`0.3::edge(A, B) :- member(A, [a, b]), member(B, [c, d]).`

It compactly encodes the facts:
`0.3::edge(a, c).`

```
0.3::edge(a, d).
0.3::edge(b, c).
0.3::edge(b, d).
```
△

Other extensions to the ProbLog language supported by the ProbLog2 system are constraints [15] and annotated disjunctions [87, 86]. While they increase the language expressivity, internally they are converted to probabilistic facts and Prolog rules. That is why we make the distinction between the *core ProbLog language* limited to probabilistic facts and Prolog rules and the *extended ProbLog language* which contains also constraints and annotated disjunctions. We discuss the extended ProbLog language in Chapter 4. In this chapter we focus on the core ProbLog language.

2.1.2 Semantics

Each probabilistic fact of a ProbLog program can be seen as an independent binary random variable – it can be either true or false. A choice of the truth value of a ground probabilistic atom is called an *atomic choice*. We can choose a probabilistic atom to be true with the probability of the fact or false with (1− the probability)[2]. The atomic choices of all probabilistic atoms (of a ProbLog program) define a *total choice*. For n probabilistic facts there are 2^n total choices. Each total choice extends to a (unique) model of the ProbLog program called a *possible world*. A ProbLog program specifies a probability distribution on possible worlds according to the Distribution Semantics [65]. Given that probabilistic atoms are considered independent random variables, we define the probability of a possible world as the product of the probabilities associated with the atomic choices.

Formally, given a ProbLog program, let $\Omega = \{\omega_1, .., \omega_N\}$ be the set of possible worlds of that program, where $N = 2^n$ and n is the number of probabilistic atoms. Given that only probabilistic atoms have probabilities we see a single possible world ω_i as the tuple (ω_i^+, ω_i^-), where ω_i^+ is the set of probabilistic atoms in ω_i which are true and ω_i^- the set of probabilistic atoms which are false according to the atomic choices. Intuitively, the union $\omega_i^+ \cup \omega_i^-$ is the set of all possible probabilistic atoms of the ProbLog program with a specific truth value assignment as defined by the corresponding total choice. The intersection $\omega_i^+ \cap \omega_i^-$ is the empty set. A ProbLog program then defines a distribution over possible worlds as given in Equation 2.1 where p_i denotes the probability of the atom a_i.

[2]The semantics of ProbLog can be generalized for atoms which are not associated with probabilities but with weights given in the form of real numbers, arithmetic expressions, Boolean functions or even data structures [35].

$$P(\omega_i) = \prod_{a_j \in \omega_i^+} p_j \prod_{a_j \in \omega_i^-} (1 - p_j) \tag{2.1}$$

The sum of the probabilities of all possible worlds associated with a ProbLog program equals one:

$$\sum_{\omega_i \in \Omega} P(\omega_i) = 1 \tag{2.2}$$

A query atom q is true in a possible world ω^q if the model expressed by the world entails q, i.e., $\omega^q \models q$. A query can be true in a set of possible worlds $\Omega^q \subseteq \Omega$. Each $\omega_i^q \in \Omega^q$ has a corresponding probability as computed by Equation 2.1. The (*success* or *marginal*) probability of q for a given ProbLog program is the sum of the probabilities of all possible worlds in which q is true:

$$P(q) = \sum_{\omega_i \in \Omega, \omega_i \models q} P(\omega_i) \tag{2.3}$$

Example 2.4. *The query* `p(a, c)` *for the ProbLog program encoding the 3-node acyclic graph (Example 2.1) is true if there is at least one path from node* `a` *to node* `c`*. The query is true in* 5 *out of the* $2^3 = 8$ *possible worlds:*

Possible World	*e(a,b)*		*e(a,c)*		*e(b, c)*		*Probability*
ω_1	*T*	*0.6*	*T*	*0.3*	*T*	*0.8*	**0.144**
ω_2	*T*	*0.6*	*T*	*0.3*	*F*	*0.2*	**0.036**
ω_3	*T*	*0.6*	*F*	*0.7*	*T*	*0.8*	**0.336**
ω_4	*T*	*0.6*	*F*	*0.7*	*F*	*0.2*	*0.084*
ω_5	*F*	*0.4*	*T*	*0.3*	*T*	*0.8*	**0.096**
ω_6	*F*	*0.4*	*T*	*0.3*	*F*	*0.2*	**0.024**
ω_7	*F*	*0.4*	*F*	*0.7*	*T*	*0.8*	*0.224*
ω_8	*F*	*0.4*	*F*	*0.7*	*F*	*0.2*	*0.056*
					$\sum =$		*1.0*

In bold font are the probabilities of possible worlds which entail the query `p(a, c)`*. Their sum, i.e., the marginal probability of the query, equals* 0.636*. The sum of the probabilities of all possible worlds equals 1 as given by Equation 2.2.* △

2.1.3 Inference and Learning Tasks

Example 2.4 shows one of the inference tasks ProbLog computes, i.e., the marginal probability of a query or the *MARG* task. Given a ProbLog program L with a set of ground derived and probabilistic atoms At and queries $Q \subseteq At$, formally the MARG task is to compute the probability each query $q \in Q$ is true for the ProbLog program L: $P(q)$. ProbLog can also compute the *conditional*

probability of a query given evidence, i.e., the *COND* task. Evidence is a set of tuples of atoms and their observed truth values. We denote evidence as $E = e$ where $E \subset At$ is the set of atoms and e the set of their truth values. The conditional probability of each query $q \in Q$ is computed as the ratio $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$. For example, given the ProbLog program encoding the 3-node acyclic graph (Example 2.1), the query `p(a, c)` and the evidence `e(a, c) =` $false$ the conditional probability $P($`p(a, c)`$|$`e(a, c) =` $false) = 0.48$. We can consider the MARG task a special case of the COND task where no evidence is given, i.e., $E = \emptyset$.

Another task ProbLog can compute is the maximum a posteriori – the *MAP* task, and its special case the most probable explanation – the *MPE* task. Consider a ProbLog program L with a set of ground atoms At, queries Q and evidence $E = e$ such that $\{Q \cup E\} \subset At$, then MAP is the task of finding the most likely atomic choices for each of the query atoms given that the evidence holds, i.e., $argmax_{\mathbf{q}} P(Q = \mathbf{q}|E = e)$, where $\mathbf{q}$ is a set of the truth values for each $q \in Q$. In the case of $\{Q \cup E\} = At$ then the task to compute is referred as the MPE task. In particular, for the MPE task ProbLog is given a program and a set of evidence atoms with their truth values, then all other ground atoms are considered queries. The solution of the MPE task, called the MPE state coincides with the possible world with the highest probability. For the ProbLog program of the 3-node acyclic graph given the evidence `e(a, b) =` $false$ the MPE state is {`e(a, c) =` $false$,`e(b, c) =` $true$}, i.e., possible world ω_7 in the table of Example 2.4. We discuss how we compute the MPE task for ProbLog programs in Chapter 4.

ProbLog is also a learning framework. It can learn the parameters of a ProbLog program given partial interpretations of that program – the *Learning From Interpretations* (*LFI*) task. A partial interpretation is a set of atoms with their observed values; it coincides with the notion of evidence. For a given ProbLog program containing a set of probabilistic facts with unknown probabilities, the LFI task computes the maximum likelihood probabilities of the probabilistic facts given evidence on some atoms. Computing the LFI task requires to consecutively perform COND inference and update the probabilities of the probabilistic facts in the initial ProbLog program, until convergence. We do not discuss it further, as the implementation of this task is based on the inference pipeline.

2.1.4 The Program Function

Equation 2.2 is a sum over products. Each product is the probability of a single possible world as given by Equation 2.1. In this section we define a more

elaborate mathematical function that captures the probability distribution of a ProbLog program. We relied on the network function presented in [10] for reasoning in the context of Bayesian networks in order to develop a function suitable for ProbLog programs. This function we called the *program function*.

Consider a ProbLog program L with At the set of all ground atoms of L – both probabilistic and derived. A possible world ω of L specifies the truth value of all atoms – the truth value of probabilistic atoms is true or false according to the atomic choices in the possible world; the truth value of a derived atom is true if the possible world entails that atom or false otherwise.

The probability of ω, computed according to Equation 2.1, can be expressed as a function of indicator and weight variables for each atom in At as follows:

$$f_\omega = \prod_{a_i \text{ is true in } \omega} \lambda_{a_i} * \theta_{a_i} \prod_{a_i \text{ is false in } \omega} \lambda_{\neg a_i} * \theta_{\neg a_i} \tag{2.4}$$

where λ_{a_i} is the indicator variable and θ_{a_i} the weight variable associated to the atom $a_i \in At$. $\lambda_{\neg a_i}$ and $\theta_{\neg a_i}$ are indicator and weight variables for the negation of the atom a_i. We denote this function as the *product function*.

For readability we omit any arguments of the product function (Equation 2.4). The arguments of a product function f_ω are the indicator and weight variables associated with the atoms in ω. That is why we use the index $_\omega$ to declare these arguments implicitly.

The indicator variables indicate whether the corresponding atom is true in the possible world. For an atom a λ_a can be equal to either 1 or 0. Logically, if $\lambda_a = 1$ then $\lambda_{\neg a} = 0$ and vice-versa. The weight variables, the θs equal the probability of the atom they correspond to. For a probabilistic atom a, θ_a is the probability of a and $\theta_{\neg a}$ is the complementary of θ_a, i.e., $\theta_{\neg a} = 1 - \theta_a$; for a derived atom a, $\theta_a = \theta_{\neg a} = 1$.

Example 2.5. *For the possible world ω_3 of Example 2.4 the corresponding product function according to Equation 2.4 is:*

$$\begin{aligned} f_{\omega_3} &= \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} \\ &= \lambda_{e^{ab}} * 0.6 * \lambda_{e^{\neg ac}} * 0.7 * \lambda_{e^{bc}} * 0.8 * \lambda_{p^{ab}} * 1 * \lambda_{p^{ac}} * 1 * \lambda_{p^{bc}} * 1 \end{aligned}$$

△

In order to cover the complete distribution of a ProbLog program L each possible world $\omega_i \in \Omega$ should be associated with a product function. We define the *program function* as the sum of these products for all possible worlds:

$$\mathcal{F}_L = \sum_{\omega_i \in \Omega} f_{\omega_i} \tag{2.5}$$

for L the ProbLog program and Ω the complete set of possible worlds of L.

Similar to the product function (Equation 2.4) we omit the arguments of the program function for sake of readability. The arguments of a program function $\mathcal{F}_L$ for a ProbLog program L with a set of atoms At are the indicator and weight variables associated with each atoms in At and its negation.

Example 2.6. *The ProbLog program encoding the 3-node acyclic graph (Example 2.1) is associated with the following program function:*

$$\begin{aligned}
\mathcal{F}_L = {} & \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{e^{ac}} * \theta_{e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} + \\
& \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{e^{ac}} * \theta_{e^{ac}} * \lambda_{\neg e^{bc}} * \theta_{\neg e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{\neg p^{bc}} * \theta_{\neg p^{bc}} + \\
& \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} + \\
& \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \theta_{\neg e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{\neg p^{ac}} * \theta_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * \theta_{\neg p^{bc}} + \\
& \lambda_{\neg e^{ab}} * \theta_{\neg e^{ab}} * \lambda_{e^{ac}} * \theta_{e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{\neg p^{ab}} * \theta_{\neg p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} + \\
& \lambda_{\neg e^{ab}} * \theta_{\neg e^{ab}} * \lambda_{e^{ac}} * \theta_{e^{ac}} * \lambda_{\neg e^{bc}} * \theta_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \theta_{\neg p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{\neg p^{bc}} * \theta_{\neg p^{bc}} + \\
& \lambda_{\neg e^{ab}} * \theta_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{\neg p^{ab}} * \theta_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \theta_{\neg p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} + \\
& \lambda_{\neg e^{ab}} * \theta_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \theta_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \theta_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \theta_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * \theta_{\neg p^{bc}} \\
\mathcal{F}_L = {} & \lambda_{e^{ab}} * \lambda_{e^{ac}} * \lambda_{e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{ac}} * \lambda_{p^{bc}} * 0.144 + \\
& \lambda_{e^{ab}} * \lambda_{e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{ac}} * \lambda_{\neg p^{bc}} * 0.036 + \\
& \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{ac}} * \lambda_{p^{bc}} * 0.336 + \\
& \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * 0.084 + \\
& \lambda_{\neg e^{ab}} * \lambda_{e^{ac}} * \lambda_{e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{p^{ac}} * \lambda_{p^{bc}} * 0.096 + \\
& \lambda_{\neg e^{ab}} * \lambda_{e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{p^{ac}} * \lambda_{\neg p^{bc}} * 0.024 + \\
& \lambda_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{p^{bc}} * 0.224 + \\
& \lambda_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * 0.056
\end{aligned}$$

△

Let $val(\mathcal{F}, \alpha)$ be the *value of a program function* $\mathcal{F}$ where all uninstantiated variables are given a value α.

The program function can be used to compute the COND and MARG tasks. To compute the MARG task for a query q we require to take into account only those products in the program function that contain λ_q. We achieve this by considering the first derivative of the program function over the variable λ_q. The mathematical operation of differentiation – the method to compute a derivative – states that for some function $g(x_1, .., x_n) = \prod_{i=1}^{n} c_i x_i$, its first derivative over the variable x_j is $\frac{\partial g}{\partial x_j} = c_j \prod_{i=1, i\neq j}^{n} c_i x_i$, $j \in \{1, .., n\}$. For a function $h(x_1, .., x_n) = \prod_{i=1, i\neq j}^{n} c_i x_i$, $j \in \{1, .., n\}$ the first derivative $\frac{\partial h}{\partial x_j} = 0$. For a function $k(x_1, .., x_n) = g(x_1, .., x_n) + h(x_1, .., x_n)$ the first derivative $\frac{\partial k}{\partial x_j} = \frac{\partial g}{\partial x_j} + \frac{\partial h}{\partial x_j} = c_j \prod_{i=1, i\neq j}^{n} c_i x_i + 0 = c_j \prod_{i=1, i\neq j}^{n} c_i x_i$. That is, the first derivative over a certain variable determines the part of the function, i.e., only the product terms, that contain the variable. In analogy to possible worlds, the first derivative of the program function over a variable λ_q corresponds to the possible worlds (of the ProbLog program) in which the atom q is true.

Computing the marginal probability of a query q then boils down to (i) computing the first derivative of the program function $\mathcal{F}$ over λ_q – $\frac{\partial \mathcal{F}}{\partial \lambda_q}$ and (ii) computing the value of the derivative when the free variables are set to 1: $val(\frac{\partial \mathcal{F}}{\partial \lambda_q}, 1)$.

Example 2.7. *For the program function $\mathcal{F}_L$ in Example 2.6 the first derivative with respect to the query $q =$* `p(a, c)`*, $\frac{\partial \mathcal{F}}{\partial \lambda_q}$ equals:*

$$\begin{aligned}
\frac{\partial \mathcal{F}_L}{\partial \lambda_{p^{ac}}} = {} & \lambda_{e^{ab}} * \lambda_{e^{ac}} * \lambda_{e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{bc}} * 0.144 + \\
& \lambda_{e^{ab}} * \lambda_{e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{p^{ab}} * \lambda_{\neg p^{bc}} * 0.036 + \\
& \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{bc}} * 0.336 + \\
& \lambda_{\neg e^{ab}} * \lambda_{e^{ac}} * \lambda_{e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{p^{bc}} * 0.096 + \\
& \lambda_{\neg e^{ab}} * \lambda_{e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{bc}} * 0.024 +
\end{aligned}$$

computing its value $val(\mathcal{F}_L, 1)$ gives the marginal probability of `p(a, c)`*:*

$$\begin{aligned}
val(\frac{\partial \mathcal{F}_L}{\partial \lambda_{p^{ac}}}, 1) &= 0.144 + 0.036 + 0.336 + 0.096 + 0.024 \\
&= 0.636
\end{aligned}$$

△

Given evidence, in order to compute the conditional probability of a query, we build the program function to contain only products which correspond to possible worlds entailing the evidence. That is, we say that the program function is consistent with the evidence.

Example 2.8. *The program function of the ProbLog program of the 3-node acyclic graph (Example 2.1) given the evidence that* `edge(a, c)` *is false is:*

$$\begin{aligned}
\mathcal{F}_L^{e^{ac}=false} = {} & \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{p^{ac}} * \theta_{p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} + \\
& \lambda_{e^{ab}} * \theta_{e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \theta_{\neg e^{bc}} * \lambda_{p^{ab}} * \theta_{p^{ab}} * \lambda_{\neg p^{ac}} * \theta_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * \theta_{\neg p^{bc}} + \\
& \lambda_{\neg e^{ab}} * \theta_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{e^{bc}} * \theta_{e^{bc}} * \lambda_{\neg p^{ab}} * \theta_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \theta_{\neg p^{ac}} * \lambda_{p^{bc}} * \theta_{p^{bc}} + \\
& \lambda_{\neg e^{ab}} * \theta_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \theta_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \theta_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \theta_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \theta_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * \theta_{\neg p^{bc}} \\
\mathcal{F}_L^{e^{ac}=false} = {} & \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{ac}} * \lambda_{p^{bc}} * 0.336 + \\
& \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * 0.084 + \\
& \lambda_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{p^{bc}} * 0.224 + \\
& \lambda_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * 0.056
\end{aligned}$$

$val(\mathcal{F}_L^{e^{ac}=false}, 1)$ is no longer equal to 1.0 *and as such does not correspond to a probability distribution. Then we need to normalize every product by* $0.7 = val(\mathcal{F}_L^{e^{ac}=false}, 1)$*, resulting in the following (normalized) function:*

$$\begin{aligned}
\mathcal{F}_L^{e^{ac}=false} = {} & \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{p^{ab}} * \lambda_{p^{ac}} * \lambda_{p^{bc}} * 0.48 + \\
& \lambda_{e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * 0.12 + \\
& \lambda_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{p^{bc}} * 0.32 + \\
& \lambda_{\neg e^{ab}} * \lambda_{\neg e^{ac}} * \lambda_{\neg e^{bc}} * \lambda_{\neg p^{ab}} * \lambda_{\neg p^{ac}} * \lambda_{\neg p^{bc}} * 0.08
\end{aligned}$$

To compute the COND task for the query `p(a, c)` *given the evidence* `e(a, c)`$=$ $false$ *we perform the same computations as in Example 2.7 on the function* $\mathcal{F}_L^{e^{ac}=false}$*:*

$$val(\frac{\partial \mathcal{F}_L^{e^{ac}=false}}{\partial \lambda_{p^{ac}}}, 1) = 0.48$$

△

Constructing a program function consistent with the evidence $\mathcal{F}_L^{E=e}$ and normalizing the θ values allows us to compute conditional probabilities as simply as computing the marginals. That is, for a query q we differentiate $\mathcal{F}_L^{E=e}$ with respect to the indicator variable λ_q and evaluate: $P(q|E=e) = val(\frac{\partial F_L^{E=e}}{\partial \lambda_q}, 1)$. For convenience, we first differentiate and then normalize:

$$P(q|E=e) = \frac{val(\frac{\partial \mathcal{F}_L^{E=e}}{\partial \lambda_q}, 1)}{val(\mathcal{F}_L^{E=e}, 1)} \tag{2.6}$$

For a set of queries the program function needs to be evaluated for each query separately. In the parameter learning setting (that is, computing the LFI task) some probabilistic facts have unknown probabilities. This reflects on the weight variables in the program function. To perform the LFI task, the program function is evaluated in order to estimate the values for these variables which maximize the value of the program function. The process is iterated until convergence.

2.2 ProbLog Inference

2.2.1 Weighted Model Counting by Knowledge Compilation

Enumerating the possible worlds of a ProbLog program in a table as shown in Example 2.4 and then marginalizing or building the program function and evaluating it for one or several queries is a straightforward approach to do probabilistic inference. Because the number of possible worlds grows exponentially with the number of probabilistic facts in a ProbLog program, these approaches are practically impossible.

In order to avoid the expensive enumeration of possible worlds the inference mechanism of ProbLog uses knowledge compilation (KC). Knowledge compilation [12] refers to a set of techniques for compiling a Boolean formula for which certain inference tasks are computationally expensive into a representation where the same tasks are tractable. That is, the complexity is shifted to the knowledge compilation step. In the context of ProbLog, knowledge compilation is used to reduce the inference task to an efficient weighted model counting. Model counting is the process of determining the number of models of a formula. Let $SAT(\varphi)$ be the set of models of the Boolean formula φ. Then the model count of φ, $\#SAT(\varphi) = \sum_{m_i \in SAT(\varphi)} 1$. The *Weighted Model Count* (WMC) of a formula φ is the sum of the weights that are associated with each model of φ: $WMC(\varphi) = \sum_{m_i \in SAT(\varphi)} w(m_i)$, where w is a weight function that associates a weight with a model. For a given ProbLog program L with a set of possible worlds Ω the WMC of a formula φ coincides with Equation 2.3 when: (i) $|\Omega| = |SAT(\varphi)|$; (ii) for each model $m_i \in SAT(\varphi)$ with weight $w(m_i)$ there is a possible world $\omega_j \in \Omega$ with probability $p(\omega_j)$ such that $w(m_i) = p(\omega_j)$; and (iii) for each possible world ω_j there is a model m_i such that $p(\omega_j) = w(m_i)$. We say that there is a bijection of the models of φ and their weights with the possible worlds of L and their probabilities.

The task of Model Counting (and also its specialization Weighted Model Counting) is in general a $\#P$-complete problem. Its importance in SAT and in

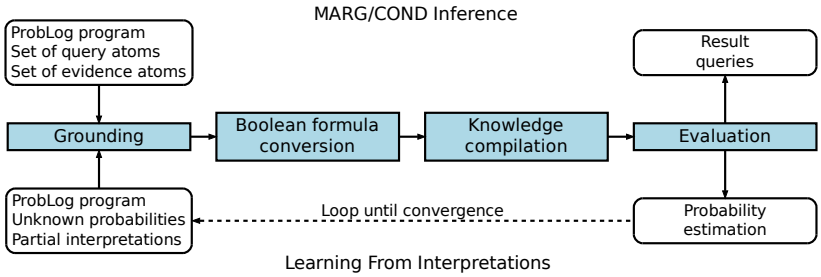


Figure 2.1: A general scheme of a ProbLog inference pipeline.

the Statistical Relational Learning (SRL) and Probabilistic Logic and Learning communities has lead to the development of efficient knowledge compilation algorithms [10] which have found their place in ProbLog. By using knowledge compilation, the actual WMC can be computed linearly in the size of the compiled formula [10, Chapter12].

2.2.2 Inference Pipeline

In order to transform a ProbLog inference task into a WMC problem an initial ProbLog program (together with a set of query and evidence atoms) needs to be compiled into a Boolean formula with special properties that allows to efficiently perform WMC.

To do so ProbLog uses a sequence of transformation steps, called an *inference pipeline*. There are four main transformation steps, i.e., *components* that compose an inference pipeline: *Grounding*, *Boolean formula conversion*, *Knowledge Compilation* and *Evaluation*. The general scheme of an inference pipeline is depicted in Figure 2.1. Figure 2.1 illustrates the main differences between MARG or COND inference and LFI. In both cases the main part of the inference pipeline is the same. For the LFI though, the input is a ProbLog program where some probabilistic facts have unknown probabilities, the inference pipeline is used to estimate these probabilities which are then used in the next iteration; the cycle of iterations is terminated once convergence of the output probabilities is reached.

The general scheme of a ProbLog pipeline is the following: the starting point is a ProbLog program with a possibly empty set of queries and a possibly empty set of evidence (depending on the inference task). During the grounding the ProbLog program is considered a purely logical program, that is, the probabilistic labels of probabilistic facts are ignored. The grounder generates

a propositional instance of that program. Second, this propositional instance is converted to an equivalent, with respect to the models, Boolean formula. Third, the Boolean formula is compiled into a *negation normal form* (NNF) with certain properties which allow efficient model counting. Finally, this NNF is converted to an *arithmetic circuit* (AC in short) – a mathematical representation of a Boolean formula, which is associated with the probabilities of the input ProbLog program and weighted model counting is performed.

Each component can use different tools or algorithms to perform the necessary transformation, as long as the input/output requirements between components are respected. For example, ProbLog1 [13] uses knowledge compilation to ROBDDs while ProbLog2 [14] uses knowledge compilation to sd-DNNFs. In order to comply with these requirements it may be the case that an intermediate data formatting is needed. For example, the Boolean formula that needs to be compiled to ROBDD or sd-DNNF needs to be formatted as a BDD script or rewritten in Conjunctive Normal Form (CNF) accordingly.

Figure 2.2 gives an overview of the different approaches that can be used to implement a component and how they can be linked in order to form an inference pipeline. Each node of the graph represents an input/output format; each edge states a transformation and points from the output to the input. Solid edges define an existing pipeline. Default pipelines are indicated by (*) for MetaProbLog/ProbLog1 and (**) for ProbLog2. Dashed edges indicate a nonexistent pipeline, i.e., pipelines that are not used in an official ProbLog release. Dashed nodes indicate intermediate data formats. The input ProbLog program may contain query and evidence atoms. Vertical arrows alongside the graph indicate the components.

In the remaining of this section we analyze each component separately.

Grounding

A naive grounding approach is to generate all possible instances of the initial ProbLog program according to all the values the variables can be bound to. Such a complete grounding may result in extremely big ground instances. Furthermore, not all ground atoms and rules are necessary for computing the probability of a query. It is more efficient with respect to the size of the grounding and the time for its generation to consider only the part of the logic program[3] that is relevant to a set of query and evidence atoms. We say that a ground logic program is *relevant* to an atom q if it contains only relevant atoms

[3]During grounding the probabilistic information, i.e., the probability label of each probabilistic fact, is ignored. That is, by *logic program* we mean the logic part of the input ProbLog program.

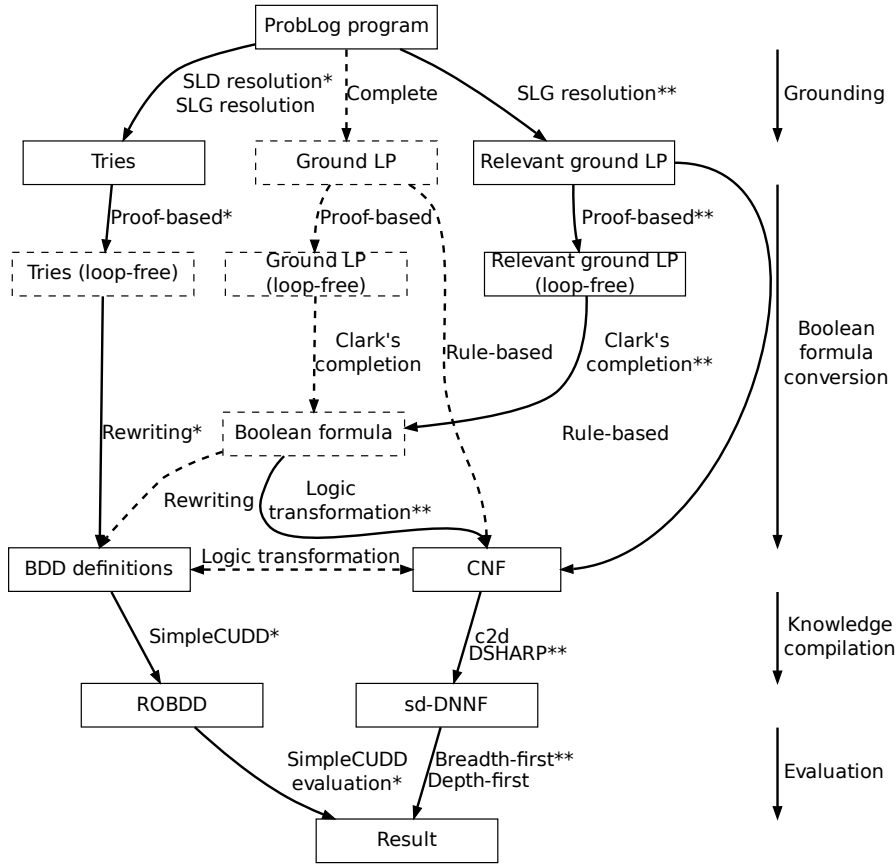
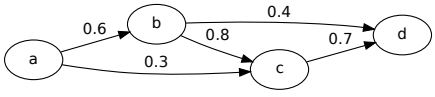


Figure 2.2: ProbLog pipelines.

and rules. An atom is relevant if it appears in some *proof* of q. A ground rule is relevant if its head is a relevant atom and its body consists of relevant atoms or negations of relevant atoms; the body should be true. Complete and relevant groundings are shown in Example 2.9.

Example 2.9 (4-node acyclic graph)**.** *Consider the following probabilistic graph and its encoding as a ProbLog program:*




```
0.6::e(a, b).  0.3::e(a, c).
0.8::e(b, c).  0.4::e(b, d).
0.7::e(c, d).
p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

Grounding the program completely and relevant to the query `p(a, c)` *results in the following grounding logic programs.*

Complete grounding:

```
0.6::e(a, b).
0.3::e(a, c).
0.8::e(b, c).
0.4::e(b, d).
0.7::e(c, d).
p(a, b):- e(a, b).
p(b, c):- e(b, c).
p(c, d):- e(c, d).
p(a, c):- e(a, c).
p(a, c):- e(a, b), p(b, c).
p(b, d):- e(b, d).
p(b, d):- e(b, c), p(c, d).
p(a, d):- e(a, b), p(b, d).
p(a, d):- e(a, c), p(c, d).
```

Relevant grounding for query `p(a, c)`*:*

```
0.6::e(a, b).
0.3::e(a, c).
0.8::e(b, c).
p(b, c):- e(b, c).
p(a, c):- e(a, c).
p(a, c):- e(a, b), p(b, c).
```

△

A possible world of the relevant ground program is a partial possible world of the complete ground program. It is safe to confine to the ground program relevant to q because the possible worlds of the relevant ground program extend to the possible worlds of the initial ProbLog program which entail the atom q. The relevant ground program captures the distribution $P(q)$ entirely. We illustrate it in Example 2.10 and refer the interested reader to [14] for more details and proof of correctness.

Example 2.10. *For each of the ground logic programs in Example 2.9 we list the sets of possible worlds in which the query* `p(a, c)` *is true. For the ProbLog program which corresponds to the complete grounding of the input program these are:*

Possible World	*e(a,b)*		*e(a,c)*		*e(b, c)*		*e(b, d)*		*e(c, d)*		*Probability*
$\omega_{1,1}$	*T*	*0.6*	*T*	*0.3*	*T*	*0.8*	*T*	*0.4*	*T*	*0.7*	*0.04032*
$\omega_{1,2}$	*T*	*0.6*	*T*	*0.3*	*T*	*0.8*	*T*	*0.4*	*F*	*0.3*	*0.01728*
$\omega_{1,3}$	*T*	*0.6*	*T*	*0.3*	*T*	*0.8*	*F*	*0.6*	*T*	*0.7*	*0.06048*
$\omega_{1,4}$	*T*	*0.6*	*T*	*0.3*	*T*	*0.8*	*F*	*0.6*	*F*	*0.3*	*0.02592*
											$\Sigma = \mathbf{0.144}$
$\omega_{2,1}$	*T*	*0.6*	*T*	*0.3*	*F*	*0.2*	*T*	*0.4*	*T*	*0.7*	*0.01008*
$\omega_{2,2}$	*T*	*0.6*	*T*	*0.3*	*F*	*0.2*	*T*	*0.4*	*F*	*0.3*	*0.00432*
$\omega_{2,3}$	*T*	*0.6*	*T*	*0.3*	*F*	*0.2*	*F*	*0.6*	*T*	*0.7*	*0.01512*
$\omega_{2,4}$	*T*	*0.6*	*T*	*0.3*	*F*	*0.2*	*F*	*0.6*	*F*	*0.3*	*0.00648*
											$\Sigma = \mathbf{0.036}$
$\omega_{3,1}$	*T*	*0.6*	*F*	*0.7*	*T*	*0.8*	*T*	*0.4*	*T*	*0.7*	*0.09408*
$\omega_{3,2}$	*T*	*0.6*	*F*	*0.7*	*T*	*0.8*	*T*	*0.4*	*F*	*0.3*	*0.04032*
$\omega_{3,3}$	*T*	*0.6*	*F*	*0.7*	*T*	*0.8*	*F*	*0.6*	*T*	*0.7*	*0.14112*
$\omega_{3,4}$	*T*	*0.6*	*F*	*0.7*	*T*	*0.8*	*F*	*0.6*	*F*	*0.3*	*0.06048*
											$\Sigma = \mathbf{0.336}$
$\omega_{4,1}$	*F*	*0.4*	*T*	*0.3*	*T*	*0.8*	*T*	*0.4*	*T*	*0.7*	*0.02688*
$\omega_{4,2}$	*F*	*0.4*	*T*	*0.3*	*T*	*0.8*	*T*	*0.4*	*F*	*0.3*	*0.01152*
$\omega_{4,3}$	*F*	*0.4*	*T*	*0.3*	*T*	*0.8*	*F*	*0.6*	*T*	*0.7*	*0.04032*
$\omega_{4,4}$	*F*	*0.4*	*T*	*0.3*	*T*	*0.8*	*F*	*0.6*	*F*	*0.3*	*0.01728*
											$\Sigma = \mathbf{0.096}$
$\omega_{5,1}$	*F*	*0.4*	*T*	*0.3*	*F*	*0.2*	*T*	*0.4*	*T*	*0.7*	*0.00672*
$\omega_{5,2}$	*F*	*0.4*	*T*	*0.3*	*F*	*0.2*	*T*	*0.4*	*F*	*0.3*	*0.00288*
$\omega_{5,3}$	*F*	*0.4*	*T*	*0.3*	*F*	*0.2*	*F*	*0.6*	*T*	*0.7*	*0.01008*
$\omega_{5,4}$	*F*	*0.4*	*T*	*0.3*	*F*	*0.2*	*F*	*0.6*	*F*	*0.3*	*0.00432*
											$\Sigma = \mathbf{0.024}$

For the relevant ground program the possible worlds in which the query is true are:

Possible World	*e(a,b)*		*e(a,c)*		*e(b, c)*		*Probability*
ω_1	*T*	*0.6*	*T*	*0.3*	*T*	*0.8*	**0.144**
ω_2	*T*	*0.6*	*T*	*0.3*	*F*	*0.2*	**0.036**
ω_3	*T*	*0.6*	*F*	*0.7*	*T*	*0.8*	**0.336**
ω_5	*F*	*0.4*	*T*	*0.3*	*T*	*0.8*	**0.096**
ω_6	*F*	*0.4*	*T*	*0.3*	*F*	*0.2*	**0.024**

One possible world of the relevant ground program corresponds to four possible worlds of the complete ground program. That is, it is a partial possible world for the complete ground program which expands to four complete possible worlds. The sum of products will then include $0.4 + 0.6 = 1.0$ *for the fact* `e(c, d)` *being true and false, and* $0.7 + 0.3 = 1.0$ *for* `e(b, d)` *being true and false.* △

To determine the relevant grounding, a natural mechanism is to use SLD resolution [38] and its extension for negative clauses SLDNF [40]. Given a query q to a logic program, SLDNF resolution builds an SLDNF tree where each node represents a set of (sub)goals that need to be proven. A subgoal is a literal – an atom or its negation. The root of the SLDNF tree is the query q[4]. In order to prove a goal g_i that unifies with the head of a rule g_i:- $sg_1, .., sg_m$ the goal g_i is substituted with the subgoals $sg_1, .., sg_m$; if g_i unifies with a fact then it is removed from the set of goals. A goal that is a negated atom succeeds

[4]The query can be a literal, a conjunction of literals or a disjunction of literals.

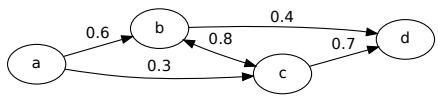
if the atom cannot be proven, i.e., all SLDNF derivations starting with the atom fail. After each substitution a child node is generated in the SLDNF tree. A successful SLDNF derivation for a query q is a depth-first trace of the SLDNF tree from the root (i.e., the query q) to an empty set of subgoals. Each successful SLDNF derivation determines one proof of q – a set of ground literals that all need to be true, i.e., a conjunction of ground literals. Naturally, all proofs to a query form a disjunction and therefore, can be represented as a Boolean formula in Disjunctive Normal Form (DNF).

An SLDNF derivation may be infinite, e.g., in case of cyclic programs. In order to avoid complications caused by cycles a suitable approach is tabling. Tabling-based approaches like OLDT resolution [81] or SLG resolution [6] are widely used in logic programming. Basically, when a tabled subgoal is called for the first time it is memoized in a table; repeated calls to that subgoal reuse the memoized entry in the table. This mechanism allows loops to be detected and avoided.

In order to determine the grounding of a ProbLog program ProbLog systems use either SLD resolution with a limited treatment of negation (details about negation follow in Section 2.2.3) or a form of SLG resolution. In particular, ProbLog1 and its descendant MetaProbLog allow the user to select whether to use tabling or not during grounding. The tabling mechanism of these systems is a variant of the YAP tabling engine [62][5] customized and optimized for the specifics of the ProbLog systems. The grounding method of ProbLog2 uses the standard YAP tabling engine. It is worth mentioning that YAP tabling has some limitations. A significant limitation for ProbLog is the undefined behavior of YAP tabling for cycles over negation.

We compare the two approaches, SLD resolution and SLG resolution, on a 4-node cyclic graph in Example 2.11.

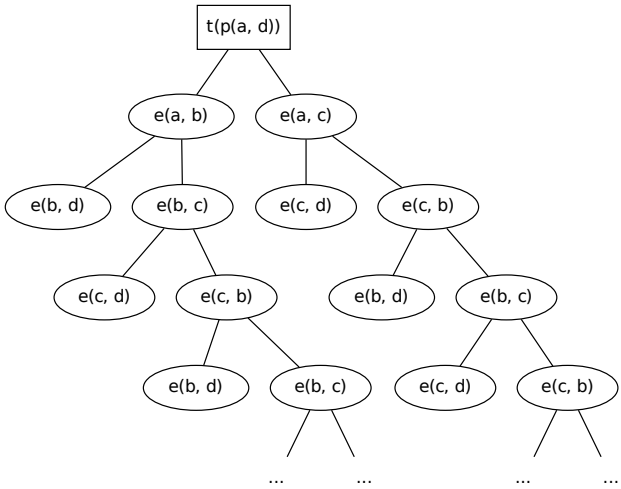
Example 2.11 (4-node cyclic graph)**.** *We extend the probabilistic graph of Example 2.9 with an edge that creates a cycle, namely* `0.8::e(c, b).`*.*



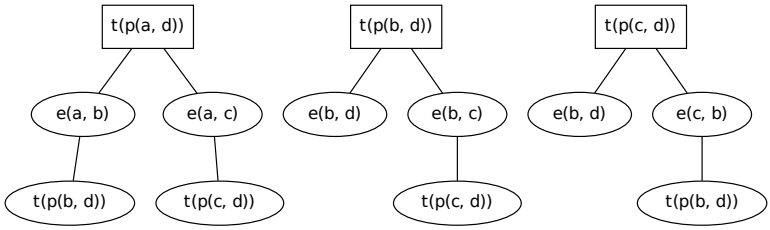

```
0.6::e(a, b).  0.3::e(a, c).
0.8::e(b, c).  0.8::e(c, b).
0.4::e(b, d).  0.7::e(c, d).
p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

Then we use the logic part of the program and SLD and SLG resolution to compute the proofs of the query `p(a, d)`*. We use tries to present the collected proofs. The SLD tree is infinite:*

[5]The YAP tabling engine, called YAPTab, implements SLG-WAM [80].



SLG resolution generates a forest of nested tries. For each tabled goal a new trie is generated. For the given program we table the predicate `p/2`*. The forest of nested tries for the query* `p(a, d)` *is:*



△

Cycles can be detected by introducing additional code to the logic program in order to store and compare intermediate results while performing SLD resolution. Such a mechanism though, can become slow and is susceptible to user errors. That is why tabling (i.e., SLG resolution) is preferable for ProbLog inference.

To collect the relevant grounding ProbLog implementations use the SLD or SLG resolution, adapted to perform extra bookkeeping of the probabilistic information.

We distinguish between two representations of the relevant grounding of a ProbLog program. ProbLog1 and MetaProbLog use the **trie data structure**

as an intermediate representation of the collected proofs: if SLG resolution is used then it is a forest of nested tries; otherwise, if SLD resolution is used, it is a single trie which corresponds to the SLD tree. Both cases are illustrated in Example 2.11. Since we focus on pipelines which use SLG resolution in the remaining of this chapter we use the term nested tries to refer to the data structure that represents the collected proofs, except if not mentioned otherwise explicitly. ProbLog2 considers the **relevant ground logic program** with respect to a set of query or evidence atoms (see Example 2.9). A relevant ground logic program can be easily extracted from the nested trie structure. However, to avoid traversing the forest of nested tries ProbLog2 employs a grounding mechanism that builds the relevant ground ProbLog program while proving a query or an evidence atom. Once a subgoal is proven it is directly added to the relevant ground LP. This may cause rules that are true but are not relevant to be included in the relevant ground LP. For example, in order to prove the body of the rule r_i :- $b_1, .., b_n$. each literal b_j needs to be proven. If b_j is proven to be true, then the relevant part for b_j is added to the relevant ground LP. If a literal b_k, $k > j$, $k \leq n$, is false then the whole body is false and none of the atoms $r_i, b_1, .., b_n$ is relevant and needs to appear in the relevant ground LP (assuming that no other rule with head r_i exists and is true and $b_1, .., b_n$ do not appear in any successful SLD derivation). Thus, the relevant part for b_j already added is redundant and does not need to appear in the relevant ground LP. The default ProbLog2 pipeline handles this issue appropriately in the next component.

The relevant ground LP can be saved in a file and then used as input for the next component, namely the Boolean formula conversion.

It is worth mentioning that probabilistic inference can be conducted directly on the collected proofs for a given query. Seeing the collected proofs as a Boolean formula in DNF is the most natural view. Each literal is associated with a probability (a literal of a derived atom has probability 1.0). Because probabilistic facts are interpreted as independent random variables (see Section 2.1) then the marginal probability of a query corresponds to the probability that the (DNF) formula is true. If all conjunctions in the DNF are disjoint, i.e., do not share any atoms, then it is simple to compute the probability from the DNF by substituting disjunctions with summations and conjunctions with multiplications and compute the value of the resulting arithmetic circuit. In the more general case, when conjunctions in the DNF share atoms, this approach is incorrect[6]. One way to reason directly on the DNF is to employ the Inclusion-Exclusion Principle but it is an inefficient method when it comes to typical ProbLog applications. In [73], we present an approximation technique that uses

[6]Some PLP systems like PRISM [66] require that users write their programs such that proofs are mutually exclusive.

DNF to compute an estimate of the MARG probability of a query. Another approximation method implemented for ProbLog is Program Sampling [34] – a Monte Carlo technique that randomly samples a logic program from a ProbLog program and tests if the query succeeds for the sampled program.

Boolean Formula Conversion

To enable the translation of the grounding into a logic form that allows efficient WMC, the forest of nested tries (in the case of ProbLog1) or the relevant ground LP (in the case of ProbLog2) is converted into a Boolean formula. This requires a semantic switch from Logic Programs (LP) semantics to First-Order Logic (FOL) semantics.

LPs use the Closed World Assumption (CWA), which states that if an atom cannot be proven to be true, it is false. In contrast, FOL has different semantics. Consider the (FOL) theory $\{q \leftarrow p\}$ which has three models: $\{\neg q, \neg p\}$, $\{q, \neg p\}$ and $\{q, p\}$. Its syntacticly equivalent LP (`q :- p.`) has only one model, namely $\{\neg q, \neg p\}$. In order to generate a Boolean formula from nested tries or from a relevant ground LP it is required to make the transition from LP semantics to FOL semantics. When the grounding does not contain cycles it suffices to take the Clark's completion of that program [40, 23, 28]. When the grounding contains cycles e.g., `p(a, c) :- e(a, b), p(b, c).` `p(b, c):-e(b, a), p(a, c).` it is proven that the Clark's completion does not result in an equivalent Boolean formula [28]. That is, we need to handle the cycles correctly so that a Boolean formula φ that is generated from a grounding of an input ProbLog program L has the following property: there is a bijection between the possible worlds of L (Ω) and the models of φ ($SAT(\varphi)$), i.e., $\Omega = SAT(\varphi)$.

To handle cyclic groundings ProbLog implementations employ one of two methods.

1. The **proof-based** approach [43] traverses the set of proofs of a query and removes proofs that contain cycles as they do not contribute to the probability. It is query-directed. It can be applied on the data structure that represents the proofs, i.e., the relevant ground LP or the nested tries, and modifies it into a cycle-free data structure. Then it can easily be rewritten as a Boolean formula. Auxiliary variables may be introduced in the process.

2. The **rule-based** approach is inherited from the field of Answer Set Programming. It rewrites a rule that produces a positive cycle to an

equivalent set of rules that are free from positive cycles [28][7]. In order to disallow cycles it introduces auxiliary variables.

Both approaches handle the cycles so that the bijection requirement is satisfied. ProbLog1 and MetaProbLog use only the proof-based approach.

Once the cycles are handled, ProbLog1 and MetaProbLog rewrite the formula encoded in the nested tries as **BDD definitions** [43]. A BDD definition is a formula with a head and a body, linked with equivalence. The body of a BDD definition contains literals and/or heads of other BDD definitions combined by conjunctions or disjunctions. The logic operators are translated to arithmetic functions – $\Longleftrightarrow$ to $=$, $\wedge$ to $*$, $\vee$ to $+$, $\veebar$ to $\#$. A BDD script is a set of BDD definitions. A BDD script is the input for the compiler used to build a ROBDD. The particular form of a BDD script enables to efficiently build a ROBDD in a bottom-up manner.

In the case of ProbLog2, once the cycles are handled the relevant ground LP is converted to a formula in **CNF**[8]. The relevant ground LP can also be rewritten to BDD definitions.

Example 2.12 shows the Clark's completion and the equivalent CNF[9] formula after applying the proof-based approach.

Example 2.12. *Consider the relevant ground LP associated with the 4-node cyclic graph from Example 2.11 – to retrieve the relevant ground LP we can traverse the forest of nested tries and collect the relevant ground atoms and rules. This program contains cycles. We use the proof-based approach on that program to remove the cycles. The result is a new ground LP with no cycles that has the same possible worlds as the initial one.*

[7]To use the rule-based Boolean formula conversion a ground ProbLog program is translated into an AnsProlog* [79] syntax.

[8]The CNF is output in a file in DIMACS format.

[9]ProbLog2 uses the DIMACS format for Boolean formula in CNF.

The cycle-free ground program is:

```
p(a,d):- e(a,c), p(c,d).
p(a,d):- e(a,b), p(b,d).
p(c,d):- e(c,d).
p(c,d):- e(c,b), aux0.
p(b,d):- e(b,d).
p(b,d):- e(b,c), p(c,d).
aux0:- e(b,d).
```

And the Clark's completion of that program:

$$\texttt{p(a,d)} \iff \texttt{aux1} \vee \texttt{aux2}$$
$$\texttt{aux1} \iff \texttt{e(a,c)} \wedge \texttt{p(c,d)}$$
$$\texttt{aux2} \iff \texttt{e(a,b)} \wedge \texttt{p(b,d)}$$
$$\texttt{p(c,d)} \iff \texttt{e(c,d)} \vee \texttt{aux3}$$
$$\texttt{p(b,d)} \iff \texttt{e(b,d)} \vee \texttt{aux4}$$
$$\texttt{aux3} \iff \texttt{e(c,b)} \wedge \texttt{aux0}$$
$$\texttt{aux4} \iff \texttt{e(b,c)} \wedge \texttt{p(c,d)}$$
$$\texttt{aux0} \iff \texttt{e(b,d)}$$

The Boolean formula in CNF is:

$$(\neg\texttt{p(a,d)} \vee \texttt{aux1} \vee \texttt{aux2}) \wedge$$
$$(\texttt{p(a,d)} \vee \neg\texttt{aux1}) \wedge$$
$$(\texttt{p(a,d)} \vee \neg\texttt{aux2}) \wedge$$
$$(\texttt{aux1} \vee \neg\texttt{e(a,c)} \vee \neg\texttt{p(c,d)}) \wedge$$
$$(\neg\texttt{aux1} \vee \texttt{e(a,c)}) \wedge$$
$$(\neg\texttt{aux1} \vee \texttt{p(c,d)}) \wedge$$
$$(\texttt{aux2} \vee \neg\texttt{e(a,b)} \vee \neg\texttt{p(b,d)}) \wedge$$
$$(\neg\texttt{aux2} \vee \texttt{e(a,b)}) \wedge$$
$$(\neg\texttt{aux2} \vee \texttt{p(b,d)}) \wedge$$
$$(\neg\texttt{p(c,d)} \vee \texttt{e(c,d)} \vee \texttt{aux3}) \wedge$$
$$(\texttt{p(c,d)} \vee \neg\texttt{e(c,d)}) \wedge$$
$$(\texttt{p(c,d)} \vee \neg\texttt{aux3}) \wedge$$
$$(\neg\texttt{p(b,d)} \vee \texttt{e(b,d)} \vee \texttt{aux4}) \wedge$$
$$(\texttt{p(b,d)} \vee \neg\texttt{e(b,d)}) \wedge$$
$$(\texttt{p(b,d)} \vee \neg\texttt{aux4}) \wedge$$
$$(\texttt{aux3} \vee \neg\texttt{e(c,b)} \vee \neg\texttt{aux0}) \wedge$$
$$(\neg\texttt{aux3} \vee \texttt{e(c,b)}) \wedge$$
$$(\neg\texttt{aux3} \vee \texttt{aux0}) \wedge$$
$$(\texttt{aux4} \vee \neg\texttt{e(b,c)} \vee \neg\texttt{p(c,d)}) \wedge$$
$$(\neg\texttt{aux4} \vee \texttt{e(b,c)}) \wedge$$
$$(\neg\texttt{aux4} \vee \texttt{p(c,d)}) \wedge$$
$$(\neg\texttt{aux0} \vee \texttt{e(b,d)}) \wedge$$
$$(\texttt{aux0} \vee \neg\texttt{e(b,d)})$$

△

BDD definitions are equivalent to a Boolean formula. As such they can be rewritten as a Boolean formula in CNF and vice versa. Example 2.13 and Example 2.14 show two cases where a formula is represented as BDD definitions and as CNF as well as the possible complications related to switching between the two forms.

Example 2.13. *For the ProbLog program of the 3-node acyclic graph (Example 2.1) and the query* `p(a, c)` *the Boolean formula associated with the completion of the relevant ground LP is:* $(p_{ac} \iff (e_{ac} \vee (e_{ab} \wedge p_{bc}))) \wedge (p_{bc} \iff e_{bc})$*, where* p_{xy} *and* e_{xy} *denote* `p(x, y)` *and* `e(x, y)` *respectively. Following are its equivalent representations as a CNF and BDD definitions where* $a0$ *stands for an auxiliary Boolean variable:*

CNF:	$(\neg p_{ac} \vee e_{ac} \vee a0) \wedge (p_{ac} \vee \neg e_{ac}) \wedge (p_{ac} \vee \neg a0) \wedge (a0 \vee \neg e_{ab} \vee \neg p_{bc}) \wedge$ $(\neg a0 \vee e_{ab}) \wedge (\neg a0 \vee p_{bc}) \wedge (p_{bc} \vee \neg e_{bc}) \wedge (\neg p_{bc} \vee e_{bc})$
BDD definitions:	p_{ac} `=` e_{ac} `+ a0` `a0 =` e_{ab} `*` p_{bc} p_{bc} `=` e_{bc}

△

Example 2.14. *A CNF can be rewritten as BDD definitions and vice-versa by a set of logical transformations. The following BDD definitions are generated from the CNF in Example 2.13 and are equivalent to the formula in Example 2.13:*

BDD definitions:	`a1 =` ~p_{ac} `+` e_{ac} `+ a0` `a2 =` p_{ac} `+` ~e_{ac} `a3 =` p_{ac} `+` ~`a0`
	`a4 = a0 +` ~e_{ab} `+` ~p_{bc} `a5 =` ~`a0 +` e_{ab} `a6 =` ~`a0 +` p_{bc}
	`a7 =` p_{bc} `+` ~e_{bc} `a8 =` ~p_{bc} `+` e_{bc}
	`a9 = a1 * a2 * a3 * a4 * a5 * a6 * a7 * a8`

△

The Clark's completion of a cycle-free logic program is a formula similar to the one in Example 2.13. This formula can easily be converted to CNF as well as in BDD definitions. Example 2.13 shows that a CNF representation of such a formula is less succinct ([12]) than the representation as BDD definitions. If though such a CNF formula is converted to BDD definitions as in Example 2.14 the BDD script blows up in size. For the overall performance of a pipeline it is crucial to avoid components which perform such a transformation. This phenomenon is discussed among others in [60]. In [14] we consider a ProbLog pipeline in which a CNF formula is transformed into BDD definitions as shown in Example 2.14, i.e., a relevant ground LP is first converted to a Boolean formula in CNF which subsequently is converted to a BDD script. Our experiments confirm that such an approach is inefficient for ProbLog inference. That is why we do not consider inference pipelines that include a transformation from CNF to BDD definitions. To the contrary, we introduce a *new pipeline* that transforms the relevant ground program directly into BDD definitions avoiding the blow up of the BDD script (see Table 2.1, pipeline $P4$).

We ought to note some differences in the implementation of the proof-based approach in the different ProbLog systems. The MetaProbLog and ProbLog2 systems use different formula preprocessing. Namely, MetaProbLog uses the recursive node merging method presented in [46, 41] and ProbLog2 uses Boolean subformulae repetition detection. Furthermore, from the nested tries the proof-based cycle handling as implemented in MetaProbLog generates another type of tries. Namely the *depth breadth* tries or DB tries in short [46]. DB tries are tries that are adapted to conveniently represent disjunctions as *breadth* nodes and conjunctions as *depth* nodes, in order to aid the extraction of the BDD script.

Knowledge Compilation and Evaluation

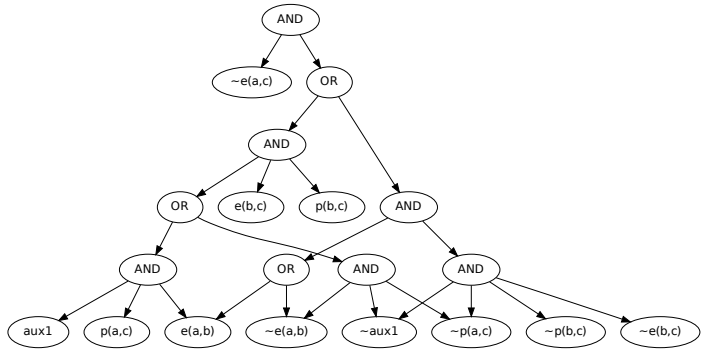
In this section we present the knowledge compilation approaches used for ProbLog inference. More details about the used compiled forms and the evaluation techniques can be found in Appendix A.

Knowledge compilation is the process in which a Boolean formula is compiled to a negation normal form (NNF) with certain properties [12]. In ProbLog's inference pipelines two target compilation languages that allow polytime weighted model counting have been exploited so far: (i) Reduced Ordered Binary Decision Diagram **ROBDD** [5] common for ProbLog1 and MetaProbLog and (ii) Smooth Deterministic Decomposable Negation Normal Form **sd-DNNF** [12] employed by ProbLog2. An ROBDD formula (or simply ROBDD) has the decision and ordering properties [12]; an sd-DNNF formula (or simply sd-DNNF) has the properties *determinism*, *decomposability* and *smooth* [12]. Using ROBDDs or sd-DNNFs one can solve various inference tasks in polytime, including WMC and model enumeration[10].

In Example 2.15 we show the sd-DNNF and in Example 2.16 the ROBDD for the same ProbLog program, query and evidence.

Example 2.15. *Given the ProbLog program of the 3-node acyclic graph (Example 2.1), the query* `p(a, c).` *and the evidence* `e(a, c)`$= false$ *ProbLog computes the conditional probability* $P($`p(a,c)`$|$`e(a,c)`$= false) = 0.48$. *First ProbLog constructs the ground LP (relevant to the query* `p(a, c).` *and the evidence* `e(a, c)`*); next it generates a CNF after using the proof-based approach to check for cycles. Then the CNF is compiled into an sd-DNNF with the* DSHARP *compiler [54]. We represent an sd-DNNF as a directed graph with AND nodes that denote conjunction, OR nodes to denote disjunction and leaf nodes that contain the literals:*

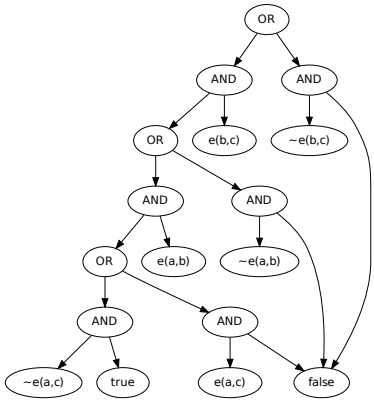
[10]The MPE inference task boils down to finding the model of the compiled formula that has the highest weight. With respect to a ProbLog program it corresponds to the possible world with the highest probability.



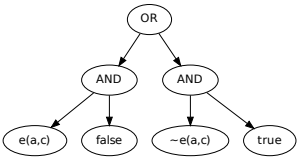
The leaf nodes correspond to derived, probabilistic and auxiliary atoms. The auxiliary atoms are created during the Boolean formula conversion. △

Example 2.16. *For the ProbLog program of the 3-node acyclic graph (Example 2.1), the query* `p(a, c).` *and the evidence* `e(a, c)`$= false$*, in order to compute the conditional probability ProbLog generates an ROBDD to compute the probability* $P($`p(a,c)`$\wedge$`e(a,c)`$= false)$ *and another ROBDD to compute the evidence –* $P($`e(a,c)`$= false)$*. In practice ProbLog generates a forest of ROBDDs that share nodes when possible. For simplicity, in the example we show the ROBDDs without node sharing. In order to compare ROBDDs to sd-DNNFs, we use a rather uncommon representation of ROBDDs – as sentences in the NNF language. We visualize such sentences as a directed graph with AND nodes to denote conjunction, OR nodes to denote disjunction and leaf nodes that contain literals,* true *or* false*. We use this representation for ROBDDs in order to show the differences and similarities with sd-DNNFs.*

The ROBDD for the conjunction of the query and the evidence `p(a,c)`$\wedge$`e(a,c)`$= false$ *is:*



The ROBDD for the evidence `e(a,c)`$= false$ *is:*



△

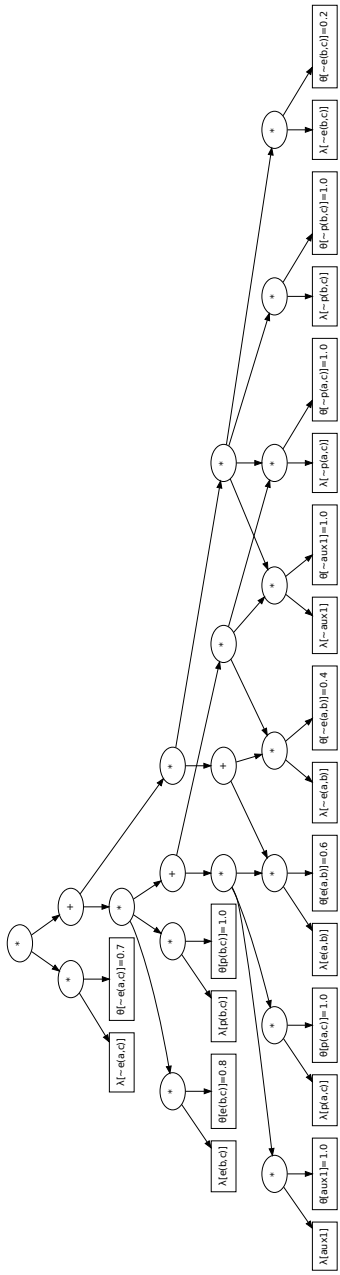
The compiled formula is then evaluated in order to compute the required task for the given query and evidence atoms – the evaluation step. During evaluation the compiled sd-DNNF is converted into an arithmetic circuit (AC) – conjunctions and disjunctions are substituted by mathematical operations and the leaf nodes are associated with indicator and weight variables (λs and θs); the weight variables are instantiated with the probability of the corresponding atom. The AC is constructed with respect to the task that needs to be solved. For the COND and MARG tasks, which boil down to weighted model counting, conjunctions are substituted by multiplications in the AC and disjunctions by summations [10, Chapter 12]. While the mathematical function represented by the AC is equivalent to the program function introduced in Section 2.1.4, an AC is a much more compact representation of that function. Evaluating the AC (with respect to the MARG or COND tasks) is done by the same approach as for the Program Function (see Section 2.1.4). The MPE task is reduced to a weighted Max-SAT [16]. In order to solve the MPE task conjunctions are substituted by multiplications in the AC, and disjunctions by *maximum* operations. The MAP task computes the most likely joint state of some query atoms given evidence. That means that we need to marginalize over the atoms which are neither queries nor evidence. The impact on the AC is that some disjunction nodes are substituted by multiplication but others are substituted by maximum. The MPE task is discussed in Chapter 4. For the remaining of this chapter we focus on the COND and MARG tasks.

In Example 2.17 we give the AC derived from the sd-DNNF shown in Example 2.15.

To compute the COND or the MARG tasks, ProbLog employs two evaluation methods for sd-DNNFs: **breadth-first** and **depth-first**[11], and one for ROBDDs. To evaluate an ROBDD its representation as a decision diagram is associated with the probabilities and traversed bottom-up. In Example 2.18 we show the ROBDDs of Example 2.16 as diagrams together with the associated probabilities.

Example 2.17. *The arithmetic circuit used to compute the conditional probability* $P($`p(a, c)`$|$`e(a, c)`$= false$ *for the ProbLog program of the 3-node acyclic graph (Example 2.1).*

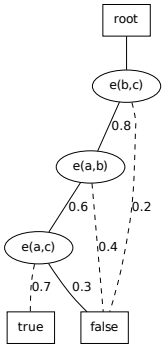
[11] To invoke one of these two options in ProbLog2 one specifies either the *fileoptimized* (default) for the breadth-first implementation or *python* for the depth-first implementation as evaluation options.


λ(au,x1)
θ(aux1)=1.0
λ(e(b,c))
θ(e(b,c))=0.8
λ(p(a,c))
θ(p(a,c))=1.0
λ(p(b,c))
θ(p(b,c))=1.0
λ(e(a,b))
θ(e(a,b))=0.6
λ(¬e(a,b))
θ(¬e(a,b))=0.4
λ(¬aux1)
θ(¬aux1)=1.0
λ(¬p(a,c))
θ(¬p(a,c))=1.0
λ(¬p(b,c))
θ(¬p(b,c))=1.0
λ(¬e(b,c))
θ(¬e(b,c))=0.2
λ(¬e(a,c))
θ(¬e(a,c))=0.7

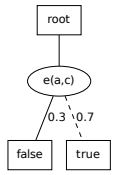
△

Example 2.18. *The ROBDDs used to compute the conditional probability* $P(\texttt{p(a, c)}|\texttt{e(a, c)}= \mathit{false}$ *for the ProbLog program of the 3-node acyclic graph (Example 2.1).*

The ROBDD for the conjunction of the query and the evidence $\texttt{p(a,c)}\wedge\texttt{e(a,c)}= \mathit{false}$ *is:*



The ROBDD for the conjunction of the evidence $\texttt{p(a,c)}\wedge\texttt{e(a,c)}= \mathit{false}$ *is:*



△

So far we described the components of the two mainstream ProbLog pipelines – ProbLog1 together with its descendant MetaProbLog and ProbLog2. The subprocesses which are used in these pipelines constitute a set of interchangeable components which may form other working pipelines. Figure 2.2 gives an overview of the possible ProbLog pipelines. The link between different components depends on the compatibility of the output of a preceding subprocess with the input requirements of the next one. For example, c2d cannot compile BDD definitions but requires CNFs. We need to adapt the output of one component so that it meets the input requirements of the next component's implementation. Earlier it was shown that some pipelines are certain to perform worse than others: pipelines with (naive) complete grounding; pipelines in which a CNF is converted to ROBDD Definitions (cf. Section 2.2.2). In addition, we prefer using SLG resolution for grounding instead of SLD resolution in order to avoid infinite proofs caused by cycles. This leaves the 14 pipelines shown in Table 2.1. We construct these pipelines combining the implementation of ProbLog2 with MetaProbLog. $P0$ to $P3$ and $P5$ to $P8$ are ProbLog2 pipelines; $P2$ is the default ProbLog2 pipeline. $P13$ is the default MetaProbLog pipeline. $P4$ and $P9$ to $P12$ are previously unexploited pipelines for ProbLog inference.

	Grounding	Boolean formula conversion	Knowledge compilation	Evaluation	New Pipeline
$P0$	SLG→Rel. gr. LP	Proof-based→CNF	c2d→sd-DNNF	Breadth-first	No
$P1$	SLG→Rel. gr. LP	Proof-based→CNF	c2d→sd-DNNF	Depth-first	No
$P2$	SLG→Rel. gr. LP	Proof-based→CNF	DSHARP→sd-DNNF	Breadth-first	No
$P3$	SLG→Rel. gr. LP	Proof-based→CNF	DSHARP→sd-DNNF	Depth-first	No
$P4$	SLG→Rel. gr. LP	Proof-based→BDD def.	SimpleCUDD→ROBDD	SimpleCUDD	**Yes**
$P5$	SLG→Rel. gr. LP	Rule-based→CNF	c2d→sd-DNNF	Breadth-first	No
$P6$	SLG→Rel. gr. LP	Rule-based→CNF	c2d→sd-DNNF	Depth-first	No
$P7$	SLG→Rel. gr. LP	Rule-based→CNF	DSHARP→sd-DNNF	Breadth-first	No
$P8$	SLG→Rel. gr. LP	Rule-based→CNF	DSHARP→sd-DNNF	Depth-first	No
$P9$	SLG→Nested tries	Proof-based→CNF	c2d→sd-DNNF	Breadth-first	**Yes**
$P10$	SLG→Nested tries	Proof-based→CNF	c2d→sd-DNNF	Depth-first	**Yes**
$P11$	SLG→Nested tries	Proof-based→CNF	DSHARP→sd-DNNF	Breadth-first	**Yes**
$P12$	SLG→Nested tries	Proof-based→CNF	DSHARP→sd-DNNF	Depth-first	**Yes**
$P13$	SLG→Nested tries	Proof-based→BDD def.	SimpleCUDD→ROBDD	SimpleCUDD	No

Table 2.1: Pipelines used in the experiments. $X \rightarrow Y$ stands for a transformation X and the output representation Y (see Fig. 2.2).

2.2.3 Support of Negation

ProbLog's semantics is based on the Least Herbrand Model (LHM) semantics[12]. For negation free logic programs the LHM is guaranteed to exist and to be unique. For programs with negation we use the well-founded model [84] as discussed in [33, 14].

The support of general negation for ProbLog is defined in [33, 41, 30] and it relies on tabling, i.e., SLG resolution and nested tries. Basically, the probability of a query `\+`q equals the probability that q fails, i.e., $P($`\+`$q) = 1 - P(q)$. Consider first an acyclic ProbLog program L. If during grounding q a negated subgoal `\+`g needs to be proven, then the goal g is proven first (or its memoized result is reused) and the result is negated. If g can be deterministically proven (that is, no probabilistic atoms are involved in the proofs of g) then its negation fails and the derivation of q that involves `\+`g fails. Otherwise, that is, g cannot be proven, then its negation succeeds. On the level of Boolean formula (see Section 2.2.2) a subformula is associated with the proofs of g and then negated to make q true.

Complications arise, though, when negation is involved in cycles, e.g., g`:- \+`g.. Such cycles lead to paradoxes where g is true if the negation of g is true and according to the well-founded semantics the truth value of g is undefined (see [41] for details). Moreover, proofs that include such cycles do not contribute to the probability. Basically, if another proof for goal g exists then the cycle can be removed; otherwise the result is undefined; from the implementation point of view, an error is raised.

2.3 Evaluation

In this section we present our results from experimenting with the 14 different ProbLog inference pipelines. Our experiments aim to determine the influence of the implementation of the different components on the performance of these pipelines. That is, which components are crucial for the overall performance.

2.3.1 Benchmarks

1. The **Alzheimer** benchmark set [13] is build from a real-world biological dataset of Alzheimer genes. The data is represented as a directed probabilistic graph with 11530 edges and 5220 nodes. We used 17

[12]The semantics of Sato [65] defines a distribution over least models.

subgraphs with increasing sizes and without duplicate edges extracted from the initial graph. We used 6 different path queries for each of the (sub)graphs. With each combination *query-graph* we associate one ProbLog program.

2. The **Balls** benchmark set, presented in [75], contains ProbLog programs encoding a game in which a player draws colorful balls (red, green and blue) from different bags one bag after another. The player can choose only one ball per bag. To be able to select from a bag the previous selections should fulfill certain conditions. The different options for a bag are encoded as annotated disjunctions [87, 50]. We use 40 different queries that compute the probability a ball is selected from a specific bag. Each query is associated with a separate ProbLog program.

3. The probabilistic **Dictionary** benchmark set ([69]) includes around 200 different words from the English language. They are linked to each according to a similarity measure expressed with a probability (probability 1.0 states that two words mean exactly the same; probability 0.0 that two words do not mean the same). The probabilities are computed according to two approaches: (i) the algorithm presented in [69] and (ii) MSR (`http://cwl-projects.cogsci.rpi.edu/msr/`). They form an *incomplete* probabilistic graph. For 30 of the words their meaning is also given. We use 65 randomly selected queries which look for the probability that two words have the same meaning even if an explicit link has not been defined. There are 7939 possible queries that involve two words for which a link is possible to exist. That is, excluding words for which it is certain (based on our data) that they are semantically unrelated. Each query is associated with one ProbLog program.

4. The probabilistic **Grid**, introduced in [14], is a special case of a probabilistic graph. We use a grid with 25×25 nodes. Each node $n_{x,y}$ is connected by a directed edge to the nodes $n_{x+1,y}$, $n_{x,y+1}$ and $n_{x+1,y+1}$. We use different queries $path(n_{1,1}, n_{x,x})$ where $x = 3, .., 25$.

5. The **Les Miserables** [36] originally is a deterministic dataset presenting the relations of the characters from the same-name novel who appeared in the same chapter. The data was shifted to a probabilistic setting by calculating the probability that two randomly selected characters will appear in the same chapter i.e., a *tie* relation. We use 68 benchmark programs each containing one query. A single query asks for the probability of a tie between two characters.

6. **Smokers** [59] is a dataset which expresses a friend network. Each person can smoke either because of stress or because he/she is *influenced* by a

Name:	Generated from:	Number of benchmark instances:	Number of programs in one instance:	Total number of programs:	Cyclic:	Inference task:
1. Alzheimer	Real-world data	6	17	102	Yes	MARG
2. Balls	Artificial data	1	40	40	No	MARG
3. Dictionary	Real-world data	1	65	65	Yes	MARG
4. Grid	Artificial data	1	25	25	No	MARG
5. Les Miserables	Real-world data	1	68	68	Yes	MARG
6. Smokers	Artificial data	1	24	24	Yes	MARG COND
7. WebKB	Real-world data	1	50	24	Yes	MARG COND

Table 2.2: Summary of the benchmarks used in our experiments. There are 6 instances from the "Alzheimer" benchmark set, therefore the total number of benchmark instances is 12.

friend who smokes. In a ProbLog program from the "Smokers" benchmark set, the `influence` relations are encoded as probabilistic facts. We use programs with an increasing number of people: from 3 until 100 which is indicative for the size of the program. One benchmark program contains multiple queries and evidence. A single query asks the probability that a person smokes. We use this set for both computing the marginal and the conditional probabilities.

7. The **WebKB** benchmark set is built upon a dataset from a collective classification domain in which university webpages are classified according to their textual content (`http://www.cs.cmu.edu/~webkb/`) [16]. All the probabilities are learned from data [23]. We use 98 programs containing different sets of queries and evidence atoms. We do both MARG and COND inference. We compute the marginal (or the conditional) probability a classification is correct (given the evidence holds).

We summarize our benchmarks in Table 2.2[13]. All benchmarks used in this work can be found at `http://people.cs.kuleuven.be/~dimitar.shterionov/benchmarks_pipelines.zip`.

The benchmark programs we use encode different directed probabilistic graphs. The graphs corresponding to the "Balls" and the "Grid" benchmarks are acyclic with a hierarchical structure. The graphs of the "Grid" have an maximum in/out degree of 3. The rest are cyclic. The "Alzheimer", "Smokers" and "WebKB" benchmarks are complex graphs with large number of cycles; the ones in the "Les Miserables" and the "Dictionary" are sparse graphs (with density < 0.0012

[13]Listed in the table are the numbers of benchmark programs we considered to use for our experiments. Our experimental conclusions are based on results for benchmark programs for which at least one pipeline terminated successfully within the timeout limit.

and < 0.0002 respectively). The benchmark programs from all but the "Balls" set are similar to the path programs shown in Example 2.1 and Example 2.9. The queries associated with these programs ask for the probability that a path between two nodes exists. A program from the "Smokers" or "WebKB" benchmark sets contains multiple queries. For the rest each program is associated with one query. The programs from the "Balls" benchmark set use annotated disjunctions [87] to encode random events with multiple outcomes. Annotated disjunctions provide an alternative and more intuitive encoding of uncertain events with multiple outcomes than probabilistic facts. ProbLog translates internally the annotated disjunctions into probabilistic graphs. Annotated disjunctions and the "Balls" benchmark set are discussed in Chapter 4.

The variety of these benchmarks ensures a close to realistic estimate of the general performance of ProbLog pipelines.

2.3.2 Experimental Setting

We tested the 14 pipelines (listed in Table 2.1) on the 7 benchmark sets discussed in the previous section. We executed the MARG task on all of the 7 benchmark sets[14] and the COND task on the last 2 sets – "Smokers" and "WebKB".

In our experiments, we measure the run times of each subprocess (grounding, conversion, compilation and evaluation) while performing the MARG or the COND task for the given query(ies) and evidence. Because the sd-DNNF compilers are non-deterministic (cf. [8, 54]), i.e., for the same CNF the compiled sd-DNNFs may differ, we run all tests 5 times and report the average run time. Previous tests with these compilers within ProbLog pipelines have shown that the average time for 5 runs gives a reliable estimate on its performance.

We run our experiments on Intel® quad-core 64-bit CPU at 2.83GHz machines with 8GBs of RAM running Ubuntu 12.04 LTS (under normal load). We set a timeout of 540 seconds for each run.

We expect that our results show how the different pipelines perform compared to each other. Then we can assess the impact that a certain component has on the overall pipeline performance. We could then identify which component(s) is crucial for a ProbLog pipeline and what the reasons are.

In Section 2.3.3 we present the run time for each pipeline on the benchmarks. Section 2.3.4 summarizes our results in tables in order to determine which are the best performing pipelines and which components are crucial. A discussion

[14] For the last two benchmark sets "Smokers" and "WebKB" in order to compute the MARG task and not the COND we ignore any evidence given in a program.

then follows in Section 2.3.5. More details about the results can be found in Appendix B.

2.3.3 Time Diagrams

We present the total runtime (the sum of the grounding, conversion, compilation and evaluation times) of each pipeline for a benchmark program executing an inference task; the lower the time is, the better. The reason to focus only on the total run time is that any change in the performance of two pipelines which share all but one component will be due to these different components. That is why, in order to get an idea of the impact of individual components we compare the result for pipelines which differ by one component. For example, comparing pipeline $P0$ to pipeline $P9$, $P1$ to $P10$, .., $P4$ to $P13$ will determine the effect of the two different grounding approaches. We present the results for the MARG task and the COND tasks separately.

In a diagram each horizontal line is associated with one program and shows the runtime of each pipeline (x-axis) executing the MARG or the COND task on that program; the colors of each line are automatically generated in accordance to the complexity of the program. The complexity is measured by the size of the dependency graph representing the ground ProbLog program. The black line parallel to the x-axis indicates the 540^{th} second, that is, the timeout. We use a logarithmic scale for the time axis (the y-axis).

MARG Inference

Figure 2.3, Figure 2.4, Figure 2.5, Figure 2.6 and Figure 2.7 show the total run time for performing MARG inference on the “Balls”, “Grid”, “Les Miserables”, “Smokers” and “WebKB” benchmark sets. The results from the “Les Miserables” benchmarks are similar to the “Alzheimer” and the “Dictionary”; although the results from the “Smokers” benchmarks are similar to the “WebKB” we show both diagrams so that later they can be compared to the results from performing COND inference shown in Figure 2.8 and Figure 2.9 respectively.

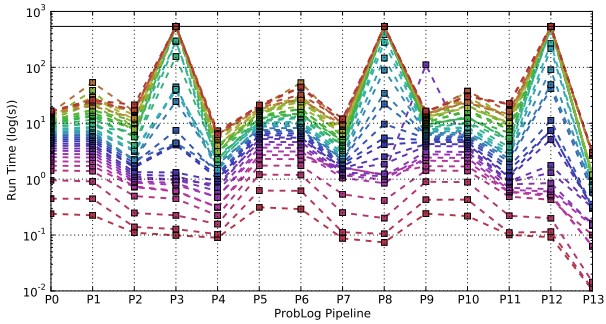


Figure 2.3: Run times for the “Balls” set.

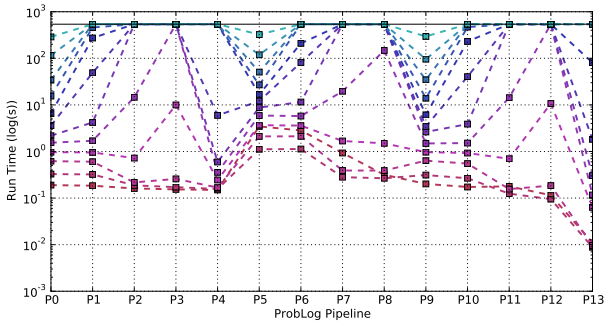


Figure 2.4: Run times for the “Grid” set.

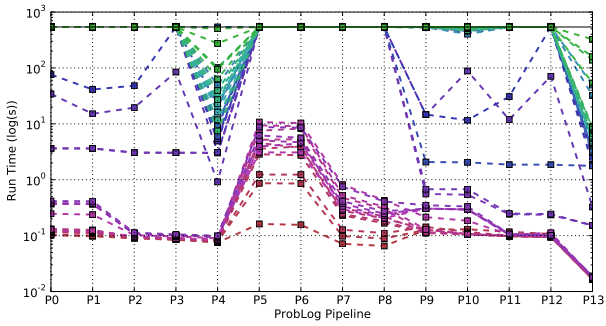


Figure 2.5: Run times for the “Les Miserables” set.

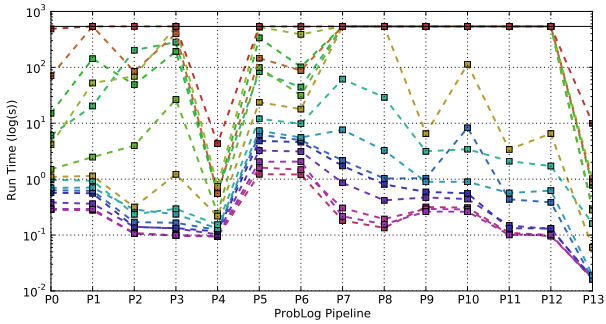


Figure 2.6: Run times for the "Smokers" set.

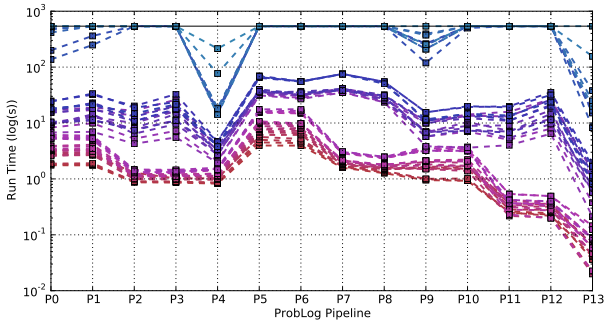


Figure 2.7: Run times for the "WebKB" set.

COND Inference

Figure 2.8 and Figure 2.9 show the run time for performing COND inference on the "Smokers" and "WebKB" benchmarks.

2.3.4 Best-performing Pipelines

We count the number of benchmark programs for which a pipeline performs best, second best and so forth. The results are shown in Table 2.3 and Table 2.5 for the *MARG* and the *COND* tasks respectively. Table 2.4 and Table 2.6 show the number of benchmark programs for which each pipeline performs best, second best and so forth when each benchmark instance (12 for the MARG task and 2 for the COND task) is given a weight of 1 and this weight is evenly

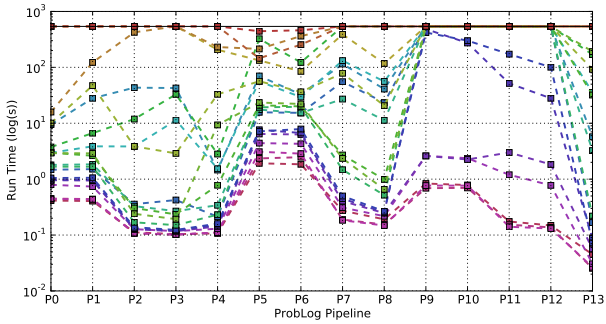


Figure 2.8: Run times for the Smokers set.

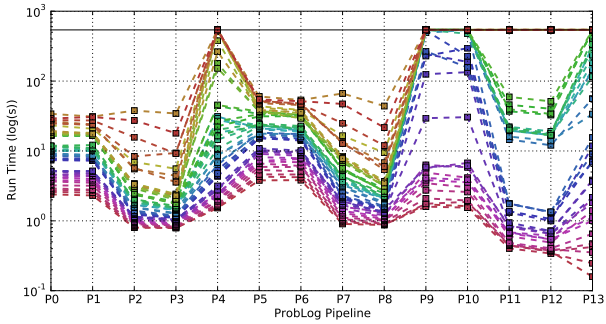


Figure 2.9: Run times for the WebKB set.

distributed over the programs within the benchmark instance, i.e., the *relative* to the total number of programs in a benchmark set successfully executed with at least one pipeline, for the MARG task and the COND task respectively. For example, pipeline $P8$ performs second best for two benchmarks – one from the "Balls" and one from the "Les Miserables" sets. There are 40 and 45 benchmarks which have been successfully executed by *at least one pipeline* in the "Balls" and "Les Miserables" sets respectively. Then we compute the relative number of programs for which $P8$ performs second best as $1/40 + 1/45 = 0.05$. The relative number shown in Table 2.4 for pipeline $P4$ performing first is 3.71. This states that in 3.71 out of 12 benchmark sets this pipeline performs best, that is in 31% of the cases $P4$ performs best. The sum of the ratios in the last rows of Table 2.4 and Table 2.6 is smaller or equal to the number of benchmark sets used for experimenting. For Table 2.4 this number is smaller or equal than 12 because for the "Alzheimer" benchmark set we used 6 instances.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	2	0	0	0	33	0	0	0	0	3	0	1	0	229
2^{th}	3	2	1	11	115	0	0	0	2	3	1	16	82	22
3^{th}	13	3	14	21	8	5	0	22	2	22	16	93	26	0
4^{th}	8	6	32	77	26	0	1	4	2	11	15	26	12	3
5^{th}	5	8	85	28	9	1	4	5	0	15	15	27	13	0
6^{th}	8	7	52	14	50	3	1	3	2	18	21	21	9	0
7^{th}	32	16	16	9	5	3	2	0	29	25	63	3	4	0
8^{th}	16	25	4	7	11	16	2	17	1	62	33	5	3	0
9^{th}	28	80	1	6	0	2	2	3	10	18	33	5	8	0
10^{th}	67	35	1	1	3	8	5	15	0	32	15	3	10	0
11^{th}	17	24	0	0	0	1	25	3	34	4	6	0	0	0
12^{th}	23	9	0	9	0	5	19	21	1	6	2	0	1	0
13^{th}	0	0	0	1	0	38	43	0	7	0	0	0	2	0
14^{th}	0	0	0	0	0	42	19	17	1	1	1	0	9	0
Sum:	222	215	206	184	260	124	123	110	91	220	221	200	179	254
Maximum possible value:				268										
Percentage:	83%	80%	77%	69%	97%	46%	46%	41%	34%	82%	82%	75%	67%	95%

Table 2.3: The number of benchmark programs for which a pipeline performs best, second best and so forth without timing out and executing the MARG task. The total number of programs shows how many programs have been executed with the corresponding pipeline and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	0.17	0	0	0	3.71	0	0	0	0	0.25	0	0.14	0	7.73
2^{th}	0.25	0.25	0.11	0.45	3.6	0	0	0	0.05	0.25	0.02	0.85	2.56	2.3
3^{th}	1.18	0.39	0.88	0.98	0.26	0.42	0	0.55	0.05	0.69	0.48	2.74	1.11	0
4^{th}	0.69	0.62	1.1	2.21	0.86	0	0.07	0.1	0.05	0.32	1.03	0.91	0.66	0.34
5^{th}	0.32	0.34	2.81	0.99	0.16	0.08	0.28	0.12	0	1.07	0.75	0.81	0.6	0
6^{th}	0.31	0.76	1.67	0.82	1.29	0.16	0.07	0.07	0.05	0.49	1.17	0.7	0.21	0
7^{th}	0.86	0.55	0.77	0.54	0.27	0.17	0.15	0	0.93	1.31	1.63	0.17	0.14	0
8^{th}	0.64	1.0	0.09	0.21	0.69	0.54	0.17	0.56	0.03	1.72	1.12	0.19	0.07	0
9^{th}	0.87	2.8	0.14	0.14	0	0.05	0.05	0.13	0.38	0.61	0.84	0.4	0.22	0
10^{th}	2.47	0.93	0.08	0.02	0.43	0.2	0.17	0.41	0	0.9	0.49	0.05	0.37	0
11^{th}	0.46	0.68	0	0	0	0.07	0.79	0.17	0.87	0.18	0.19	0	0	0
12^{th}	0.65	0.21	0	0.22	0	0.26	0.53	0.61	0.08	0.24	0.09	0	0.03	0
13^{th}	0	0	0	0.08	0	1.11	1.21	0	0.22	0	0	0	0.05	0
14^{th}	0	0	0	0	0	1.19	0.57	0.48	0.03	0.03	0.07	0	0.28	0
Sum	8.87	8.53	7.65	6.66	11.27	4.25	4.06	3.2	2.74	8.06	7.88	6.96	6.3	10.37
Maximum possible value:				12.00										
Percentage	74%	71%	64%	56%	94%	35%	34%	27%	23%	67%	66%	58%	53%	86%

Table 2.4: The number of benchmark programs relative to the total number of benchmarks in a set for which a pipeline performs best, second best and so forth without timing out and executing the MARG task. The total number of programs shows how many programs have been executed with the corresponding pipeline relative to the total number of programs in the benchmark set, and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	1	1	0	35	3	2	1	0	1	0	0	0	13	11
2^{th}	3	1	32	9	1	0	2	0	4	0	0	10	4	2
3^{th}	2	1	14	10	3	2	0	0	25	0	0	4	2	3
4^{th}	2	2	11	7	9	0	0	28	5	0	0	2	0	0
5^{th}	5	25	6	2	3	0	2	1	14	0	0	3	3	2
6^{th}	22	6	1	0	1	0	4	18	9	0	0	3	0	1
7^{th}	5	5	0	1	13	1	16	8	4	0	0	0	5	6
8^{th}	2	5	1	0	13	18	6	4	1	0	1	1	8	4
9^{th}	6	9	0	0	7	9	4	1	1	1	6	9	1	0
10^{th}	9	2	0	0	0	5	9	4	0	7	2	2	3	6
11^{th}	2	6	0	0	4	7	7	0	0	4	5	4	2	0
12^{th}	6	2	0	0	0	7	0	0	0	5	3	3	0	12
13^{th}	0	0	0	0	0	3	14	0	0	4	6	0	0	0
14^{th}	0	0	0	0	0	14	3	0	0	5	3	0	0	0
Sum:	65	65	65	64	57	68	68	64	64	26	26	41	41	47
Maximum possible value:				68										
Percentage:	96%	96%	96%	94%	84%	100%	100%	94%	94%	38%	38%	60%	60%	69%

Table 2.5: The number of benchmark programs for which a pipeline performs best, second best and so forth without timing out and executing the COND task. The total number of programs shows how many programs have been executed with the corresponding pipeline and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	0.05	0.02	0	0.88	0.15	0.1	0.05	0	0.02	0	0	0	0.27	0.46
2^{th}	0.12	0.05	0.81	0.39	0.05	0	0.1	0	0.11	0	0	0.21	0.08	0.07
3^{th}	0.1	0.05	0.52	0.21	0.15	0.1	0	0	0.55	0	0	0.08	0.04	0.09
4^{th}	0.07	0.1	0.29	0.17	0.45	0	0	0.61	0.16	0	0	0.04	0	0
5^{th}	0.13	0.61	0.12	0.1	0.09	0	0.1	0.05	0.44	0	0	0.06	0.15	0.04
6^{th}	0.49	0.15	0.05	0	0.02	0	0.17	0.55	0.22	0	0	0.15	0	0.05
7^{th}	0.19	0.22	0	0.05	0.3	0.02	0.36	0.17	0.2	0	0	0	0.1	0.18
8^{th}	0.1	0.16	0.05	0	0.27	0.46	0.21	0.17	0.05	0	0.02	0.02	0.2	0.08
9^{th}	0.15	0.27	0	0	0.15	0.39	0.11	0.05	0.05	0.02	0.12	0.19	0.05	0
10^{th}	0.27	0.04	0	0	0	0.13	0.25	0.2	0	0.15	0.07	0.07	0.06	0.21
11^{th}	0.04	0.12	0	0	0.08	0.15	0.15	0	0	0.14	0.19	0.08	0.1	0
12^{th}	0.12	0.04	0	0	0	0.15	0	0	0	0.19	0.09	0.15	0	0.25
13^{th}	0	0	0	0	0	0.09	0.41	0	0	0.08	0.18	0	0	0
14^{th}	0	0	0	0	0	0.41	0.09	0	0	0.16	0.06	0	0	0
Sum:	1.83	1.83	1.84	1.8	1.71	2	2	1.8	1.8	0.74	0.73	1.05	1.05	1.43
Maximum possible value:				2.00										
Percentage:	92%	92%	92%	90%	86%	100%	100%	90%	90%	37%	37%	53%	53%	72%

Table 2.6: The relative number of benchmark programs to the total number of benchmarks in a set for which a pipeline performs best, second best and so forth without timing out and executing the COND task. The total number of programs shows how many programs have been executed with the corresponding pipeline relative to the total number of programs in the benchmark set, and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	46	53	62	84	8	144	145	158	177	48	47	68	89	14
Total (relative):	3.14	3.48	4.35	5.34	0.72	7.76	7.94	8.8	9.28	3.95	4.12	5.04	5.71	1.64

Table 2.7: Total and relative (to the total number of benchmarks in a set) number of benchmark programs for which MARG inference times out. The lower the better.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	3	3	3	4	11	0	0	4	4	42	42	27	27	21
Total (relative):	0.15	0.15	0.15	0.2	0.29	0.0	0.0	0.2	0.2	1.25	1.25	0.94	0.94	0.55

Table 2.8: Total and relative (to the total number of benchmarks in a set) number of benchmark programs for which COND inference times out. The lower the better.

In Table 2.7 and Table 2.8 we summarize our timeout results for the MARG and COND inference tasks respectively. We present the total number of timeouts for each pipeline as well as the number of timeouts relative to the total number of benchmarks for which at least one pipeline has succeeded.

2.3.5 Discussion

To determine the influence of the different components on the overall performance we compare the run times of pipelines which differ by only one component. For example, pipeline $P0$ differs from pipeline $P9$ by the grounding component – $P0$ uses the ProbLog2 grounder to determine a relevant ground LP, while $P9$ the MetaProbLog grounder to nested tries.

We discuss the results from our experiments with the MARG task separately from the COND task. This is because computing the conditional probabilities in MetaProbLog (whose components we use to build other pipelines) differs from how conditional probabilities are computed in ProbLog2. The difference is in whether the truth values given to the evidence atoms are exploited.

MARG Inference

Grounding Comparing pipelines $P0, .., P4$ to $P9, .., P13$ in Figure 2.3 to Figure 2.7 shows that grounding to a relevant ground LP and grounding to nested tries have similar impacts on the performance. The grounding mechanism of ProbLog2 that we use to generate the relevant ground LP (see Section 2.2.2)

suffers from one drawback. This drawback may influence the next component, namely the Boolean formula conversion. The relevant grounding of successful subgoals that participate in a body of a clause is included in the ground program even if the body is false. This phenomenon is illustrated in Example 2.19.

Example 2.19. *Consider the ProbLog program of Example 2.9 and the query* `p(a, c).`*:*

```
0.6::e(a, b).   0.3::e(a, c).   0.8::e(b, c).
0.4::e(b, d).   0.7::e(c, d).
p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

The set of relevant ground atoms and clauses (i.e., the relevant ground LP) required to compute the probability of the query is:

```
0.6::e(a, b).
0.3::e(a, c).
0.8::e(b, c).
p(a, b):- e(a, b).
p(b, c):- e(b, c).
p(a, c):- e(a, c).
p(a, c):- e(a, b), p(b, c).
```

The relevant ground LP as computed by the grounding component is:

```
0.6::e(a, b).
0.3::e(a, c).
0.8::e(b, c).
0.4::e(b, d).
0.7::e(c, d).
p(a, b):- e(a, b).
p(b, c):- e(b, c).
p(a, c):- e(a, c).
p(a, c):- e(a, b), p(b, c).
```

The probabilistic facts `0.4::e(b, d).` *and* `0.7::e(c, d).` *are used in failing proofs; they are redundant and should be omitted from the relevant ground LP. This is another case where not all of the information in the ground program is relevant to the query similar to Example 2.9 and Example 2.10. Such a ground program may introduce complications for the next components.* △

The relevant ground LP in Example 2.19 contains an extra set of ground atoms (`0.4::e(b, d).` and `0.7::e(c, d).`) which do not contribute to the probability computation. The relevant ground LP associated with each benchmark from the "Grid" set contains unnecessary atoms as in Example 2.19. To ensure minimal relevant ground LP one solution is to employ a second traversal in order to remove subgoals that participate in failing derivations. While such optimization is unnecessary in the scope of the default ProbLog2 pipeline ($P2$) because the proof-based Boolean formula conversion implemented for ProbLog2 handles the excessive knowledge appropriately, it influences the rule-based Boolean formula conversion method.

Boolean Formula Conversion When comparing pipelines $P0, .., P3$, to $P5, .., P8$ we observe that the Boolean formula conversion has a strong impact on the performance. By itself the time for conversion (not shown on the diagrams) is not significant but it is the output Boolean formula that strongly influences the next components in the inference pipeline – knowledge compilation and evaluation. Knowledge compilation is computationally the most expensive task. The proof-based approach generates Boolean formulae that are easier to compile (with the c2d, DSHARP or SimpleCUDD tools), than the rule-based approach (compare $P0, .., P3$, to $P5, .., P8$ in Figure 2.5, Figure 2.6 and Figure 2.7).

Table 2.3 and Table 2.4 show that the pipelines that use rule-based conversion never perform best, that is, never 1^{st} but also not 2^{nd}. The timeout results in Table 2.7 show that pipelines using the proof-based conversion timeout 42% (5.34 for pipeline $P3$ and 9.28 for $P8$) to 59%[15] (3.14 for pipeline $P0$ and 7.76 for $P5$) less than pipelines using the rule-based method.

The input for the conversion is the grounding, represented either as a relevant ground LP (pipelines $P0$ to $P8$) or as nested tries (pipelines $P9$ to $P13$). As noted earlier, the relevant ground LP may contain additional ground atoms and clauses. The proof-based approach bypasses this issue by performing a query-directed search on the relevant ground LP in order to collect all proofs of a query. It starts by searching from a clause whose head unifies with the query. Once it finds such a clause it searches for clauses or atoms to proof the body and so forth. This traversal ignores any clauses or atoms that are not relevant to a query, such as the redundant clauses and atoms added by the grounder. The rule-based approach though, does build a Boolean formula in CNF from the relevant ground LP one clause after another. The extra ground atoms and clauses in the relevant ground LP are also considered in building the Boolean formula. The number of literals and the number of clauses in the formulae generated by this approach are often significantly larger compared to the proof-based approach. The extra information in the relevant ground LP though, does *not* have a high impact on the total performance. This is obvious from the results for the "Grid" benchmarks (Figure 2.4) where the relevant ground LP has extra knowledge similar to the program in Example 2.19.

For the effectiveness of the conversion of major importance is the presence of cycles in the grounding. We notice (Figure 2.3 and Figure 2.4) that pipelines using the rule-based conversion handle the acyclic graphs from the "Balls" and the "Grid" benchmark sets equally well or even better than some of the pipelines using the proof-based conversion. This is because the conversion does not need

[15] We use the relative number of timeouts rather than the total number of timeouts in order to determine a more general interval.

to handle any cycles and the rule-based conversion simply needs to traverse the relevant ground LP and rewrite it as a Boolean formulae.

These results show that the Boolean formula conversion is crucial for the inference pipeline.

Knowledge Compilation and Evaluation Knowledge Compilation is the computationally most expensive task in a ProbLog inference pipeline. We consider two target compilation languages: sd-DNNFs and ROBDDs. Our experiments show that even though sd-DNNFs are at least as succinct as ROBDDs [12], employing ROBDDs in a ProbLog pipeline results in lower run times and better scalability. Compare the run times of pipeline $P4$ to pipelines $P0$, .., $P3$ and $P13$ to $P9$, .., $P12$ on the diagrams in Figure 2.3, Figure 2.4, Figure 2.5, Figure 2.6 and Figure 2.7; in Table 2.7 the values for pipelines $P4$ and $P13$ are the lowest: 8 and 14 for the total number of timeouts and 0.72 and 1.64 for relevant number of timeouts, accordingly; Table 2.3 and Table 2.4 show that these pipelines $P4$ and $P13$ are the best performing ones.

ROBDDs allow polytime Boolean transformations, i.e., bounded conjunction, bounded disjunction and negation [12]. Therefore, compiling to ROBDDs can be performed in an efficient bottom-up manner. The size of an ROBDD strongly depends on the order in which variables are processed. Dynamic variable reordering [63] allows the transformation of ROBDDs during the compilation stage when new variables are added. Variable reordering is NP-complete [5]. Using heuristics it aims at an optimal size of the ROBDD. The input BDD script should not necessarily be in CNF form in contrast to compilation to sd-DNNFs. Then a BDD script can be substantially smaller than a CNF encoding the same Boolean formula.

In the case of knowledge compilation to sd-DNNFs a pipeline which uses c2d shows better scalability compared to one with DSHARP but is slower for the less complex problems. Compare, for example, the run times of pipeline $P0$ to pipeline $P2$ and $P9$ to $P11$ on the diagrams in Figure 2.3, Figure 2.4, Figure 2.5, Figure 2.6 and Figure 2.7 for the less complex and the more complex problems[16]. Furthermore, the breadth-first evaluation approach is in general preferable to the depth-first approach (compare $P0$ to $P1$ or $P11$ to $P12$), despite that in the case of the "Balls" benchmarks this evaluation method performs poorly (see $P3$, $P8$ and $P12$ in Figure 2.3). The reason is the structure of the graph associated with the relevant ground LP – low out degree, i.e., 9, long paths from the root to the nodes.

The bottom-up compilation, the dynamic reordering and the succinct

[16]Recall that the ordering on the y-axis is according to the size of the dependency graph associated with the grounding of the benchmark program (see Section 2.3.3).

representation of the Boolean formula as a BDD script are the main factors for ProbLog pipelines with ROBDDs to perform faster than those with sd-DNNFs for the MARG task.

Underlying Implementation We ought to comment also on the implementation of these algorithms. The default MetaProbLog pipeline ($P13$) is implemented in YAP Prolog except for the SimpleCUDD, which is written in C/C++. The ProbLog2 pipelines ($P0$ to $P3$ and $P5$ to $P8$) use a YAP Prolog implementation of the grounder; the rule-based conversion, DSHARP and c2d are implemented in C/C++; the proof-based conversion, both evaluation approaches and the wrapper that binds all components together are written in Python3.

Pipeline $P4$ is based on a ProbLog2 pipeline – it generates a BDD script directly from the relevant ground LP and then uses SimpleCUDD in a Python3 wrapper to compile this script into an ROBDD and evaluate. Pipelines $P9$ to $P12$ use the MetaProbLog default implementation and a Python3 script to invoke compilation to sd-DNNF and evaluation.

Using Python as a wrapper is a better solution to constructing ProbLog pipelines as it is more flexible and more modular than Prolog. But for implementing the different components may be slower.

COND Inference

The conditional probability of a query q given evidence $E = e$ is computed as the ratio $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$ (see Section 2.1.2). First both the nominator and denominator need to be computed separately. Then their division gives the final result. MetaProbLog and ProbLog2 use different approaches when it comes to computing the conditional probabilities. In particular, there are differences regarding the grounding to nested tries and compiling to ROBDDs compared to grounding to a relevant ground LP and knowledge compilation to s-DDNNFs.

Grounding We notice from comparing pipelines $P0$ to $P9$, $P1$ to $P10$, $P2$ to $P11$, $P3$ to $P12$ and $P4$ to $P13$ in Figure 2.8 and Figure 2.9 that grounding to nested tries has a negative effect on the overall performance as compared to grounding to a relevant ground LP, despite the drawback mentioned earlier. The former grounding method uses the following approach: (i) for a query q and evidence $E = e$ a new query $q^{E=e} = q \wedge E = e$ is created; (ii) $q^{E=e}$ and the atoms in E are proven in order to determine the relevant grounding (stored as nested tries). In the latter case, a query q and the atoms in E are used separately

and not in a conjunction to determine the relevant ground program. Although the two grounding approaches generate different groundings (not only the representation but also the ground atoms and clauses may differ), their impact on the next component (and therefore on the overall performance) is limited. That of the evidence atoms and their predetermined values is more substantial. The truth values of the evidence play a significant role for knowledge compilation.

Boolean Formula Conversion The Boolean formula is built by using either the proof-based or the rule-based method. In the case of pipelines $P0$ to $P9$ the Boolean formula (either represented as a CNF or as a BDD script) is augmented with clauses to state the truth values for the evidence atoms. They often help the knowledge compilation as they may prune parts of the compiled circuit.

Knowledge Compilation and Evaluation From the time diagrams for the MARG (Figure 2.6 and Figure 2.7) and the COND inference (Figure 2.8 and Figure 2.9) we see that in the case of COND inference the pipelines with knowledge compilation to sd-DNNFs perform better than pipelines with knowledge compilation to ROBDDs (compare pipelines $P0$ to $P3$ and $P5$ to $P9$). This contrasts with the results from MARG inference. There are three main reasons: (i) the evidence atoms and their truth values are used during knowledge compilation to sd-DNNFs to optimize the sd-DNNFs; (ii) the number of queries – sd-DNNFs are compiled from one CNF while compilation to ROBDDs generates a forest of ROBDDs for each query (in practice for each $q \wedge E = e$ and $E = e$); (iii) in case the conjunction $q \wedge E = e$ is false ($P(q \wedge E = e) = 0.0$ and therefore $P(q|E = e) = 0.0$) compilation to ROBDDs will still compile the necessary ROBDD to compute the probability, thus it will perform unnecessary operations (this is observed for the "WebKB" benchmark programs where a lot of the queries are false given that the evidence holds). The decreased performance due to compilation to ROBDDs is also confirmed by the timeout results in Table 2.8.

2.4 Conclusions and Future Work

In this chapter we presented the syntax, semantics and inference mechanism of ProbLog. Then we described in detail different implementations of ProbLog inference pipelines and analyzed their performance on 7 benchmark sets. Through our analysis we determined that the implementation of the Boolean formula conversion component has a crucial impact on the overall performance of the inference pipeline for both MARG and COND tasks – the basic ProbLog inference tasks. We showed that in most of the cases pipelines that use a *proof-based* Boolean formula conversion, *knowledge compilation to sd-DNNF with c2d* and the *breadth-first evaluation* approach and pipelines that use *proof-based*

conversion and *knowledge compilation to ROBDDs* perform better than the rest.

We also showed that the inference task to be computed has an effect on the performance of an inference pipeline. Namely, some pipelines that perform very well on the MARG task are less effective on the COND task. We determined that the reason is the way evidence is used. That is, for the COND task it is crucial how the evidence is handled. For the MARG task, pipelines that use *knowledge compilation to ROBDDs* are preferable; for the COND task pipelines that use *knowledge compilation to sd-DNNF* and *breadth-first evaluation* outperform the others.

Our future goals revolve around optimizing the Boolean formula so that the cost for knowledge compilation can be reduced. We present one optimization technique in Chapter 3. Also we want to improve the method to handle evidence for knowledge compilation with ROBDDs. Furthermore, one of the newly introduced pipelines, namely pipeline $P4$, that combines the grounding of ProbLog2 with the knowledge compilation and evaluation of MetaProbLog via a direct conversion of the (cycle-free) relevant ground LP to BDD definitions shows very promising results. To determine its actual place among the different ProbLog implementations we plan to further evaluate its performance on all inference and learning tasks supported by ProbLog.

Chapter 3

Compaction of Boolean Formulae for Probabilistic Inference

Probabilistic inference is a computationally expensive task. Modern Probabilistic Logic and Learning systems, such as ProbLog, use Knowledge Compilation [12] to reduce inference to a Weighted Model Counting problem that can be solved efficiently. Knowledge compilation encompasses a set of techniques to convert a Boolean formulae for which some inference tasks are computationally expensive into a representation where the same tasks are easy to execute. Knowledge compilation itself is a #P-complete [83] problem and in practice it is the bottleneck in probabilistic inference pipelines.

A ProbLog pipeline is a sequence of transformation steps (or components) that, given an initial ProbLog program and a set of queries and (possibly empty) evidence (i) compute a weighted Boolean formula that captures the probability distribution of the initial ProbLog program and (ii) use this formula to calculate the required probabilities. ProbLog uses knowledge compilation in (i) in order to generate a Boolean formula in Negation Normal Form (NNF) with properties that allow to perform (ii) efficiently.

The efficiency of knowledge compilation strongly depends on the input Boolean formula. In Chapter 2 we showed that two different Boolean formula that are logically equivalent have different impact on the performance of the knowledge compilation component and consequently of the whole system. Motivated to improve the performance of ProbLog inference pipelines we devised a method

to optimize Boolean formulae prior to knowledge compilation. Our method identifies seven Boolean subformulae patterns that can be detected and used to re-write Boolean formulae in a more compact formula, i.e., containing less Boolean variables than the original one. Our method preserves a problem-specific equivalence. In the case of ProbLog the initial and the compacted Boolean formulae are equivalent with respect to their Weighted Model Count (WMC).

While we implemented our approach in the scope of ProbLog and used common ProbLog problems to evaluate its effectiveness, it is more general and any application using Boolean formulae to represent knowledge could benefit from it. The experimental results show that in general, our compaction method improves knowledge compilation and therefore enables solving ProbLog programs previously unsolvable.

The contribution of this work is two-fold. First, the main focus has been the design and the implementation of our compaction algorithm. Second, we introduced the AND-OR graph data structure as a common data structure for the intermediate results of a ProbLog pipeline. The AND-OR graph data structure facilitates our method for compaction of Boolean formulae.

In this chapter we present our approach, the underlying implementation and the experimental results. In Section 3.1 we introduce necessary background information and acknowledge related work; Section 3.2 describes our patterns and their compaction; Section 3.3 illustrates our detection and compaction algorithms; in Section 3.4 we present our experimental results in the scope of ProbLog inference; we conclude in Section 3.5. In Appendix C we show detailed experimental results – the number and type of patterns detected on our benchmarks.

3.1 Background

3.1.1 Relevant Grounding of ProbLog Programs

A ProbLog inference pipeline (Chapter 2) is a sequence of transformation steps that convert an input ProbLog program together with a set of query and evidence atoms into a Boolean formula in negation normal form (NNF) with special properties to allow efficient Weighted Model Counting (WMC). There are four transformation steps, also called components – *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. Each component takes as input the output of the previous one and produces input for the next component.

The first component of a ProbLog pipeline is the *grounding* component. Given a ProbLog program L, a set of queries[1] and (possibly empty) evidence, ProbLog uses SLD [38] or SLG [6] resolution on the logical part of L, that is, ignoring the label of probabilistic facts, in order to determine a propositional instance of the initial ProbLog program relevant to the query and evidence atoms. Using SLG resolution instead of SLD resolution enables grounding cyclic programs without introducing additional code. Different representations of this propositional instance are considered: ProbLog1 [13, 32] and its descendant MetaProbLog [45, 43] use a forest of nested tries[2], while ProbLog2 [16, 14] uses a relevant ground logic program [14].

A trie is a tree-like data structure common for compact dictionary representation. In the context of logic programming tries are used to store proofs in a less memory-demanding structure. In particular, tries allow storing the common prefixes of proofs in a compact way reducing the consumed memory. Each depth-first traversal of a trie corresponds to one proof, i.e., a conjunction of literals. Nested tries may contain nodes that refer to other tries, that is, a nested trie is a forest of tries. One proof is then distributed among the different tries. A depth-first traversal of a trie corresponds to a different partition (a conjunction of literals) of a proof; the conjunction of all such partitions is one proof.

The relevant ground logic program is a logic program (LP) that contains only ground atoms and ground rules that are relevant to a query or an evidence atom. An atom is relevant to a query q if it appears in some proof of q; a rule is relevant to q if its head is a relevant atom and its body contains relevant atoms or negations of relevant atoms. In that perspective a relevant ground LP is equivalent to a forest of nested tries. The grounder of ProbLog2 implements an SLG-based approach to store in a LP each ground atom or rule that is proven.

3.1.2 AND-OR Graphs

We represent Boolean formulae as AND-OR graphs. An AND-OR graph is a directed graph composed by AND and OR nodes. An AND node indicates that all child nodes must be true, while an OR node indicates that at least one of the

[1] Not all inference or learning tasks need the explicit definition of queries. For example, in Chapter 2 we mention the MPE inference task, discussed in detail in Chapter 4. For MPE inference all atoms of the initial ProbLog program for which evidence is not given are considered *queries*. In practice, when solving the MPE task the user does not specify the queries explicitly; they are inferred by ProbLog during grounding.

[2] In case SLD resolution is used instead of SLG resolution, the forest of nested tries is reduced to a single nested trie. We focus on the more general representation – the forest of nested tries.

child nodes must be true. An AND-OR graph is a suitable representation for a ground logic program relative to a query q. The different clauses ($q_{i\in 1..m}$:- $r_{i,1}, ..., r_{i,n}$.) of the predicate q are processed as follows: for each clause q_i all literals $r_{i,j}$ in the body are grouped as children of an AND node. The different AND nodes then are grouped as children of an OR node labeled with q. Next, each literal $r_{i,j}$ is treated as a new query. An AND-OR graph of a query has the following characteristics: cycles that appear in the logic program also appear in the AND-OR graph; for each subgoal g there is only one OR node; an OR-node has multiple parents if the subgoal is repeated and goals proven as facts are represented by special OR nodes without children, called terminal nodes. Terminal nodes can hold additional information depending on the given application. In the scope of ProbLog we label a terminal node with the tuple $\{f, p_f\}$, where f is a probabilistic atom and p_f its probability. Each node in an AND-OR graph appears at most once and can have multiple parents and child nodes. Although AND nodes have the same label ($\wedge$ or AND) each of them is associated with the body of a specific rules. That is why we also state that each AND node appears only once in the graph.

The edge from a child node to a parent node states that the parent depends on the child node. Therefore an AND-OR graph can represent Boolean formulae that are not in normal form[3].

Definition 3.1. *An **AND-OR graph** for a query q is a directed graph $G = (V_{and}, V_{or}, V_{term}, E)$ with V_{and} a set of AND nodes, V_{or} a set of labeled OR nodes, $V_{term} \subset V_{or}$ a set of terminal nodes, $V_{nonterm} = V_{or} \setminus V_{term}$ and $E \subseteq R$ a set of directed edges, where $R = (V_{and} \times V_{or}) \cup (V_{nonterm} \times V_{and}) \cup (V_{nonterm} \times V_{or})$. The root of the graph is an OR node labeled with q.*

Example 3.1. *Consider the ProbLog program:*

```
0.6::e(a, b).     0.3::e(a, d).     0.8::e(b, c).
0.7::e(c, d).     0.4::e(d, f).     0.4::e(d, e).
0.2::e(e, f).

p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

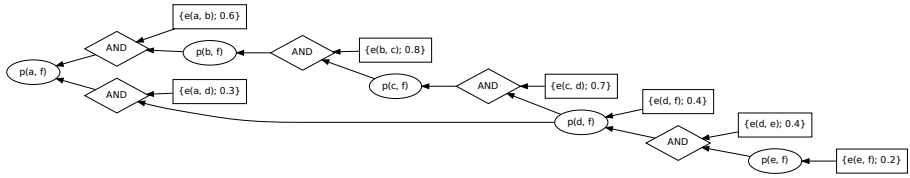
Given the query `p(a, f).` *the relevant ground LP is:*

```
e(a, b).        p(a, f):- e(a, b), p(b, f).
e(a, d).        p(a, f):- e(a, d), p(d, f).
```

[3]A Boolean formula is in normal form if it contains only conjunctions, disjunctions and negation.

```
e(b, c).        p(b, f):- e(b, c), p(c, f).
e(c, d).        p(c, f):- e(c, d), p(d, f).
e(d, f).        p(d, f):- e(d, f).
e(d, e).        p(d, f):- e(d, e), p(e, f).
e(e, f).        p(e, f):- e(e, f).
```

The AND-OR graph that corresponds to that program is:



Ellipses depict OR nodes, diamonds AND nodes and rectangles terminal nodes. OR nodes are labeled with the goal they prove. △

In the graph of Example 3.1 there is one OR node labeled with `p(e, f)` that depends on one child only – the terminal node labeled with `{e(e, f); 0.2}`. Note, that there is no intermediate AND node between these two nodes while the body of each rule needs to be associated with an AND node, as stated earlier. Such an AND node is, though, redundant and can be safely omitted. During the AND-OR graph construction we do not introduce AND nodes for ground rules that contain only one literal in their bodies.

3.1.3 Related Work

Rewriting a Boolean formula to improve the performance of knowledge compilation in the scope of ProbLog had first been investigated in [46]. The authors of [46] show that feeding a rewritten Boolean formulae instead of a non optimized one reduces the operations needed by the knowledge compilation step and consequently the knowledge compilation run time. The work we present in this chapter, focuses on optimizing even further the Boolean formulae and works in parallel with these Boolean formulae rewrites. Boolean formulae rewriting, in the scope of assessing the Probability of a Sum-of-Products, has been investigated also in [60].

Detecting regularities such as AND/OR-Clusters on a Boolean formula in normal form (i.e., DNF or CNF), has been investigated in [44, 41]. For a Boolean formula, an AND-/OR- Cluster is a conjunction/disjunction of literals such that if any literal that participates in the cluster appears in another subformulae then the rest of the literals of the cluster also appear in that subformula and form a conjunction/disjunction. Our approach performs similar to [44, 41]

transformations on an AND-OR graph instead of a Boolean formula in normal form but overcomes the practical limitations of that approach. The most important limitation is that ProbLog using tabling as presented in [43] would generate a Boolean formula that is not in normal form. The cost to convert this formula to a normal form would be high and unwanted. The use of AND-OR graphs overcomes these limitations and allows detecting AND/OR-Clusters in any Boolean formulae. Completeness of detecting AND/OR-Clusters in Boolean formulae is proven in [44, 41].

Hintsanen [25] argues that structural properties are important for finding the most reliable subgraph. He calculates the probability of subgraphs connecting two nodes and searching for the subgraph with the maximum probability. The paper identifies as a special case the series-parallel subgraphs for which they can compute the probability polynomially. These series-parallel subgraphs have similarities with the AND/OR-Clusters.

Our work is also similar to [39] which presents a preprocessing of propositional formulae to optimize model counting. Their approach optimizes CNF Boolean formulae by using seven preprocessing methods. Similar to our work, some of their preprocessing methods maintain equivalence and others not. In contrast to our approach some of their methods increase the size of the Boolean formulae. They show that in some cases redundant information may improve knowledge compilation which we confirm in our experiments. Moreover in Section 3.4.4 we identify some particular cases that depend on the target knowledge compilation language, input format and Boolean formula patterns, where compacting the Boolean formula may slow down knowledge compilation. We consider this an interesting research direction that deserves further investigation.

There exist several other related works from other fields such as variable ordering approaches for BDDs [56, 55] or preprocessing methods used in SAT solving [1, 2].

3.2 Compactable Patterns

We identify 7 different patterns that appear in AND-OR graphs and use them to compact the graph. Our compactions aim at reducing the number of Boolean variables of the formula represented by the AND-OR graph. These patterns fall into two types: one type that retains equivalence with the input Boolean formulae and a second type that reduces the number of Boolean variables contained in the formulae. The latter type patterns correspond to AND/OR clusters [44, 41]. While the second type of transformations do not preserve the equivalence directly, we ensure equivalence with an application specific problem by a special calculation for the introduced representative Boolean variable. For

ProbLog inference, which is our motivating application, we need to maintain the WMC. This requires to calculate the probability of the representative Boolean variable. The proof of correctness for compacting AND-OR clusters appears in [44, 41]. The correctness of the other patterns follows from standard Boolean formula transformations that preserve equivalence.

The detected patterns and their compacted form are illustrated in Table 3.1.

1. **Single Variable**: an OR node A and a terminal node B, such that A depends only on B. **Compaction**: node A and the edge from B to A are deleted. The edges starting from A now start from B.

2. **Single Branch I**: a node A, an OR node B and an AND node C, such that B depends only on C and A depends on B. **Compaction**: if node A is an OR node then node B and the edge from C to B are deleted. A new edge from C to A is created. If node A is an AND node then nodes B and C are deleted together with the edge from C to B. All children of C are connected to A.

3. **Single Branch II**: two OR nodes A and B, such that A depends on B and no other node depends on B. **Compaction**: node B and the edge from B to A are deleted. All children of B are connected to A.

4. **Minimal Proof**: An OR node A, two AND nodes B_1 with a set of children Ch_{B_1} and B_2 with a set of children Ch_{B_2} such that $Ch_{B_1} \subseteq Ch_{B_2}$. **Compaction**: Node B_2 and all edges from the children nodes in Ch_{B_2} to B_2 are deleted. The edge from B_2 to A is deleted as well.

5. **AND-Cluster**: An AND node A, a set of nodes $Ch'_A \subseteq Ch_A$, where Ch_A are all terminal nodes which A depends on, such that $Ch'_A = Ch_A \setminus \{C | \exists B, B \neq A, B \text{ depends on } C\}$. **Compaction**: all terminal nodes $C_i \in Ch'_A$ are deleted, together with the edges from C_i to A. A new terminal node C_t is created together with an edge from C_t to A. A joint probability $p_t = \prod_{C_i \in Ch'_A} p_i$, where C_i is a terminal node with probabilistic label p_i is calculated. The probabilistic label p_t is attached to node C_t.

6. **OR-Cluster I**: an OR node A, a set of nodes $Ch'_A \subseteq Ch_A$, where Ch_A are all terminal nodes which A depends on, such that $Ch'_A = Ch_A \setminus \{C | \exists B, B \neq A, B \text{ depends on } C\}$. **Compaction**: All terminal nodes $C_i \in Ch'_A$ are deleted, together with the edges from C_i to A. A new terminal node C_t is created together with an edge from C_t to A. A joint probability p_t is calculated as $p_t = (..((p_1 * (1-p_2) + p_2) * (1-p_3) + p_3)..+p_n)$, where p_i is probabilistic part of the label of $C_i \in Ch'_A$, $i = 1..|Ch'_A|$. The probabilistic label p_t is attached to node C_t.

7. **OR-Cluster II**: An OR node A, that depends on n AND nodes $B_1...B_n$ that each has exactly one different terminal child node $Ch_1...Ch_n$ and all the other children nodes (denoted as node C) are common. **Compaction**: All AND nodes $B_1...B_n$ and all terminal nodes $Ch_1...Ch_n$ are deleted. A new terminal node Ch is created. A joint probability p_t is calculated as $p_t = (..((p_1 * (1-p_2)+p_2) * (1-p_3)+p_3)..+p_n)$, where p_i is probabilistic part of the label of $Ch_i, i = 1..n$. The probabilistic label p_t is attached to node Ch. A new AND node B that contains Ch, C is created, finally, an edge from B to A is created.

Patterns 1 to 4 maintain the logic equivalence with the initial Boolean formula. The compaction of patterns 5 to 7 removes Boolean variables and introduces a new Boolean variable to represent them. These compactions do not directly maintain the logic equivalence of the Boolean formulae. Application specific problems require a special calculation for the newly introduced representative Boolean variable. For correct ProbLog inference we need to maintain the WMC. That requires to calculate the probability of the representative Boolean variable using the equations shown in Table 3.1. Proof of correctness for these compactions appears in [44, 41].

Pattern 1 originates from a rule with one goal in the body. Example 3.1 shows that for such rules we do not introduce additional AND nodes. AND nodes are auxiliary nodes used to represent the conjunction of the body literals of a rule. We avoid introducing unnecessary nodes to the graph by ignoring AND nodes for rules with one body literal. We cannot, though, apply a similar simplification during the AND-OR graph construction for OR nodes. It is because they correspond to the actual heads of rules and contain the information about derived ProbLog atoms.

3.3 Algorithm

Our algorithm iterates over patterns 1 to 6 in the order presented in Table 3.1. As soon as a pattern is detected the corresponding compaction is applied. According to the order we choose the detection and compaction of one pattern allows the detection and compaction of the next one in the same iteration. This ensures the minimum number of iterations required to compact a graph. By default our algorithm terminates when no more patterns can be detected. In practice, we can choose in advance the number of iterations.

Our algorithm does neither detect nor compact pattern 7 because this pattern may correspond to complex subgraphs with unreasonably high detection cost.

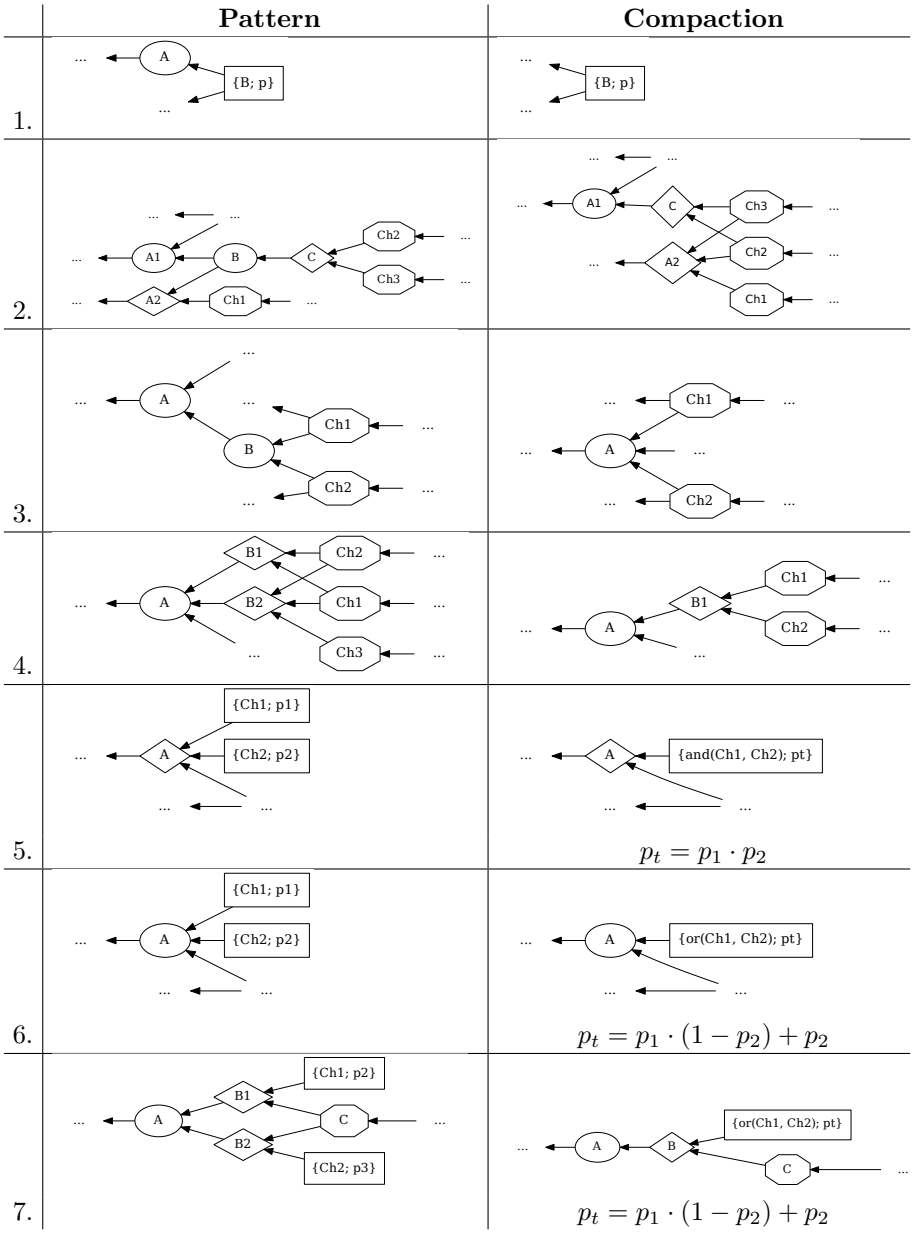


Table 3.1: AND-OR graph patterns and the compacting transformations. We depict an AND-OR graph with ellipses for OR nodes, diamonds for AND nodes and rectangles for terminal nodes. OR nodes are labeled with the goal they prove. In the context of ProbLog terminal nodes have attached probabilities. We denote with "..." multiple possible nodes to/from which exists an edge. With octagons we represent nodes that can be of any type (terminal, AND or OR).

We implemented our detection/compaction algorithm as a stand-alone Prolog program independently from any ProbLog system. We give the pseudo code for our detection algorithm in Algorithm 1 and for the compaction algorithm the pseudo code is given in Algorithm 2. We incorporated it in two different implementations of ProbLog: MetaProbLog [45, 43] and ProbLog2 [16, 14].

Algorithm 1: The 6 pattern detection algorithm.

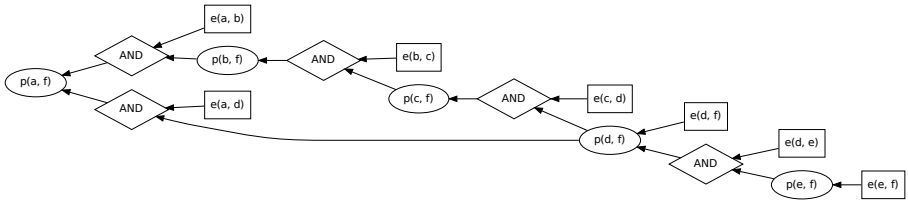
Data: An AND-OR graph
Result: Detected Node, Nodes to be compacted
detect_single_variable(NodeA, Terminal) ←
    or_edge(NodeA, Terminal),
    terminal_node(Terminal, _),
    ∄ and_edge(NodeA, _),
    ∄ (or_edge(NodeA, Any), Terminal ≠ Any).
detect_single_branch1(NodeB, NodeC) ←
    or_edge(NodeB, NodeC),
    and_edge(NodeC, _),
    ∄ and_edge(NodeB, _),
    ∄ (or_edge(NodeB, Any), NodeC ≠ Any).
detect_single_branch2(NodeA, NodeB) ←
    or_edge(NodeA, NodeB),
    or_edge(NodeB, _),
    ∄ and_edge(_, NodeB),
    ∄ (or_edge(Any, NodeB), NodeA ≠ Any).
detect_minimal_proof(NodeB2) ←
    or_edge(NodeA, NodeB1),
    and_edge(NodeB1, _),
    or_edge(NodeA, NodeB2),
    and_edge(NodeB2, _),
    NodeB1 ≠ NodeB2,
    all(Child, and_edge(NodeB1, Child), ChildrenB1),
    all(Child, and_edge(NodeB2, Child), ChildrenB2),
    ChildrenB1 ⊆ ChildrenB2.
detect_and_cluster(NodeA, RefinedChildren) ←
    and_edge(NodeA, _),
    all(Terminal, (
        and_edge(NodeA, Terminal),
        terminal_node(Terminal, _),
        ∄ or_edge(_, Terminal)
    ), Children),
    get_all_and_edge_sets(ChildSets),
    refine_cluster(ChildSets, Children, RefinedChildren),
    RefinedChildren ≠ ∅.
detect_or_cluster1(NodeA, RefinedChildren) ←
    or_edge(NodeA, _),
    all(Terminal, (
        or_edge(NodeA, Terminal),
        terminal_node(Terminal, _),
        ∄ and_edge(_, Terminal)
    ), Children),
    get_all_or_edge_sets(ChildSets),
    refine_cluster(ChildSets, Children, RefinedChildren),
    RefinedChildren ≠ ∅.
refine_cluster([], RefinedChildren, RefinedChildren).
refine_cluster([Set|ChildSets], Children, RefinedChildren) ←
    NewChildren = Set ∧ Children,
    refine_cluster(ChildSets, NewChildren, RefinedChildren).

Algorithm 2: The 6 pattern compaction algorithm.

Data: An AND-OR graph, Detected Nodes, Nodes to be compacted
Result: A compacted AND-OR graph
compact_single_variable(NodeA, Terminal) ←
  each(or_edge(Parent, NodeA),
    (retract(or_edge(Parent, NodeA)),
    assert(or_edge(Parent, Terminal))),
  retract(or_edge(NodeA, Terminal)).
compact_single_variable(NodeA, Terminal) ←
  each(and_edge(Parent, NodeA),
    (retract(and_edge(Parent, NodeA)),
    assert(and_edge(Parent, Terminal))),
  retract(or_edge(NodeA, Terminal)).
compact_single_branch1(NodeB, NodeC) ←
  each(or_edge(NodeA, NodeB),
    (retract(or_edge(NodeA, NodeB)), assert(or_edge(NodeA, NodeC))),
  retract(or_edge(NodeB, NodeC)).
compact_single_branch1(NodeB, NodeC) ←
  each(and_edge(NodeA, NodeB), retract(or_edge(NodeA, NodeB))),
  each(and_edge(NodeC, Child), assert(or_edge(NodeA, Child))),
  retract(or_edge(NodeB, NodeC)).
compact_single_branch2(NodeA, NodeB) ←
  each(or_edge(NodeB, Child),
    (retract(or_edge(NodeB, Child)), assert(or_edge(NodeA, Child))),
  retract(NodeA, NodeB).
compact_minimal_proof(NodeB2) ←
  each(and_edge(NodeB2, Child), retract(NodeB2, Child)),
  retract(or_edge(_, NodeB2)).
compact_and_cluster(NodeA, RefinedChildren) ←
  calculate_and_probability(RefinedChildren, Pnew),
  each(Child ∈ RefinedChildren,
    (retract(and_edge(NodeA, Child)), retract(terminal_node(Child, _)))),
  assert(terminal(and(RefinedChildren), Pnew)),
  assert(and_edge(NodeA, and(RefinedChildren))).
compact_or_cluster(NodeA, RefinedChildren) ←
  calculate_or_probability(RefinedChildren, Pnew),
  each(Child ∈ RefinedChildren,
    (retract(or_edge(NodeA, Child)), retract(terminal_node(Child, _)))),
  assert(terminal(or(RefinedChildren), Pnew)),
  assert(or_edge(NodeA, and(RefinedChildren))).

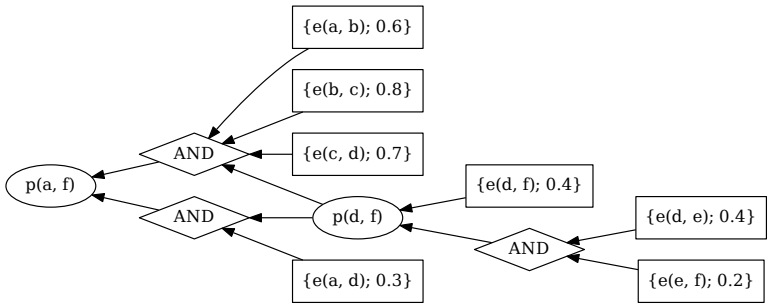
Example 3.2. *We apply our compaction algorithm on the graph in Example 3.1.*

Initial graph:

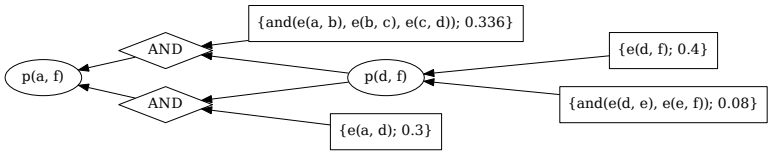


In the 1^{st} *iteration it detects 1* **Single Variable** *of* $p(e, f)$ *and 2* **Single Branch I** *of* $p(b, f)$ *and* $p(c, f)$ *resulting in Graph 1 in the following table; and 2 AND-Clusters resulting in Graph 2.*

Graph 1

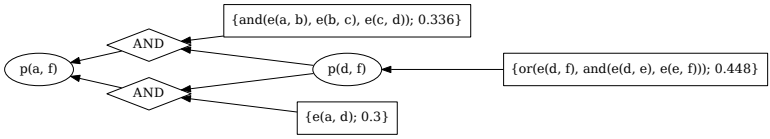


Graph 2

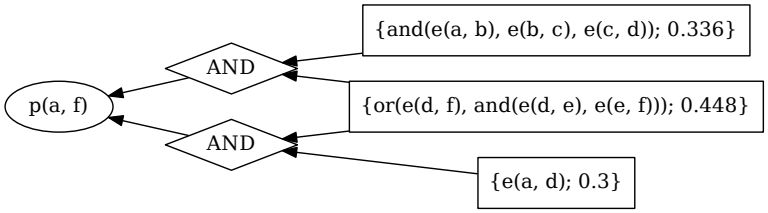


In the 2^{nd} *iteration 1* **OR-Cluster I** *and 1* **Single Variable** *of* $p(d, f)$ *are detected resulting in Graph 3 and Graph 4 accordingly.*

Graph 3



Graph 4 – final graph.



The final AND-OR graph forms 1 **OR-Cluster II** *pattern. If we detected and compacted OR-Cluster II patterns, it would enable a final AND-Cluster compaction to fully compact the AND-OR graph into a single terminal node containing the probability of the query.* △

3.3.1 Analysis

Completeness: As noted earlier our algorithm does neither detect nor compact pattern 7. We are also confident that there exist more patterns which we do not consider. Thus, AND-OR graphs which include at least one of these patterns will not be fully compacted. That is, our algorithm does not ensure optimal graph compaction.

Complexity: Our implementation uses indexed terms stored in the internal database of the YAP Prolog to represent the AND-OR graph. Finding an edge in the graph takes logarithmic time, inserting an edge or deleting an edge (after finding it) is performed in constant time. That is why, compacting a detected pattern is very efficient ($O(N)$ with N the number of edges affected). Detecting a pattern is computationally expensive and deserves further analysis.

For an arbitrary AND-OR graph G we denote with N_{or} the number of *OR edges*, with N_{and} the number of *AND edges* and with N_{term} the number of terminal nodes. We assume that a node always contains N_{term} children; this is a high upper bound assumption but does not affect the complexity class.

Patterns 1 to 3 have similar worst case complexity. Algorithm 1 performs similar operations in order to verify that an OR edge participates in one of the patterns. For detecting all Patterns 1 in an AND-OR graph it needs to check each OR edge from a terminal node B to a node A. To verify that node A is forming a Pattern 1 with node B it checks that no other OR/AND edge exists to a second node different than the terminal node B. Complexity: $O(log(N_{or}) + log(N_{and}))$ for verifying one pattern; $O(N_{or} \cdot (log(N_{or}) + log(N_{and})))$ to detect all patterns. Similarly, for Patterns 2 and 3.

To detect Pattern 4 the algorithm needs to check whether each two OR edges with the same parent are connected with two different nodes that have AND edges and form two sets of children nodes with the one being a subset of the other. The collection of the children and set comparisons are made in linear time to the number of children. Complexity to detect all patterns: $O(N_{or} \cdot (log(N_{or}) + log(N_{and}) + N_{term}))$.

For detecting all AND-Clusters we need to first collect for each AND edge all its children that are terminal nodes and not contained in OR edges, thus creating N_{and} sets of children. Then, we need for each of these sets to refine it with all the other sets in order to create a common subset. All the set operations are linear to the size of the largest set. Collecting all the nodes takes $O(N_{and} \cdot N_{term})$. Refining a candidate AND-Cluster takes $O(N_{and} \cdot N_{term})$. Complexity of detecting all patterns: $O(N_{and}^2 \cdot N_{term})$.

OR-Cluster I. The analysis of the complexity of detecting an OR-Cluster I

is similar to the one for AND-Clusters. Detecting OR-Clusters considers OR edges. Therefore, the complexity bound is for detecting all OR-Clusters I is $O(N_{or}^2 \cdot N_{term})$.

OR-Cluster II. This pattern is not implemented in our code. Here we give the theoretical complexity for the simple case (size 2) depicted in Table 3.1. We need to check all OR edges ($O(N_{or})$) and see that they have two child nodes that are parent nodes in two different AND edges. According to our implementation for the other patterns this can be done in linear time for each of the two children. Then with a set operator we can find the common children of these nodes, and need to verify that it is exactly one child ($O(N_{or} + N_{and})$). Then, we need to search all the graph (all OR and AND edges) for the child not to appear anywhere else, again with $O(N_{or} + N_{and})$ complexity. Finally, we need to verify that the terminal nodes involved in the pattern do not have other parents, with complexity $O(N_{term})$. Theoretical complexity[4] to detect all OR-Clusters II size 2: $O(N_{or} \cdot N_{term} \cdot (N_{or} + N_{and}))$. In practice this complexity bound includes the size of the cluster (2) and is $O(2^3 \cdot N_{or} \cdot N_{term} \cdot (N_{or} + N_{and}))$. For size k we estimate a theoretical complexity $O(k^3 \cdot N_{or} \cdot N_{term} \cdot (N_{or} + N_{and}))$. Furthermore, the general support of the OR-Cluster II pattern is computationally very expensive, because in each iteration our algorithm would need to search for OR-Cluster II with size $k = N_{or}$ down to $k = 2$.

3.4 Compacting ProbLog Programs

3.4.1 Employing Pattern Detection and Compaction in ProbLog

Section 3.1.1 presents the general scheme of a ProbLog inference pipeline – grounding, Boolean formula conversion, knowledge compilation and evaluation. In Chapter 2 we discussed different approaches to implement each component. Their combination resulted in 14 ProbLog pipelines. To experiment with our compaction method we consider 6 of these pipelines. They differ in (i) representation of the grounding output and the Boolean formulae: nested tries [43, 41] and BDD script [42, 41] or a relevant ground logic program and CNF; (ii) the ways of preprocessing the Boolean formulae in the Boolean formula conversion algorithm: proof-based Boolean formula conversion with recursive node merging [46, 41] or proof-based Boolean formula conversion with Boolean subformulae repetition detection; and (iii) the knowledge compilation method: compilation to ROBDDs with SimpleCUDD [42] or compilation to sd-DNNF with the compilers c2d [8] or DSHARP [54].

[4]This complexity bounds may vary depending on the actual algorithm design and implementation.

Pipeline	Grounding representation	Cycle handling	Boolean formulae representation[3]	Compilation language	Pipeline index
ProbLog2/ROBDD	Rel. gr. LP	Proof-Based[1]	AND-OR→BDD script	ROBDD	$P4$
ProbLog2/sd-DNNF (c2d)	Rel. gr. LP	Proof-Based[1]	AND-OR→CNF	sd-DNNF	$P0$
ProbLog2/sd-DNNF (DSHARP)	Rel. gr. LP	Proof-Based[1]	AND-OR→CNF	sd-DNNF	$P2$
MetaProbLog/ROBDD	Nested Tries	Proof-Based[2]	BDD script	ROBDD	$P13$
MetaProbLog/sd-DNNF (c2d)	Nested Tries	Proof-Based[2]	BDD script→CNF	sd-DNNF	$P9$
MetaProbLog/sd-DNNF (DSHARP)	Nested Tries	Proof-Based[2]	BDD script→CNF	sd-DNNF	$P11$

Table 3.2: ProbLog pipelines used in experiments. [1]Proof-based cycle handling with Boolean subformulae repetition detection. [2]Proof-based cycle handling with recursive node merging. [3] Shows also any intermediate representation used before the Boolean formula conversion.

We have chosen these particular pipelines because: (i) we want to test the effect of our compaction approach on all knowledge compilation methods considered for ProbLog inference – knowledge compilation to sd-DNNF with the DSHARP and the c2d compilers and to ROBDDs with SimpleCUDD; (ii) two of these pipelines are the default MetaProbLog and ProbLog2 pipelines and one is the pipeline which combines the grounding and the proof-based Boolean formula conversion of ProbLog2 with the knowledge compilation and evaluation of MetaProbLog and performs very well on our benchmarks (see Chapter 2) and (iii) by default ProbLog2 pipelines use AND-OR graphs as intermediate data structure to represent the relevant ground LP prior to Boolean formula conversion and a forest of nested tries is easily converted into an AND-OR graph. In contrast, a Boolean formula in CNF generated by the rule-based Boolean formula conversion method is often difficult to converted into an AND-OR graph and its optimization with our method does not lead to a substantial compaction. In Section 3.4.4 we present details and a more elaborate discussion about why we do not consider pipelines employing the rule-based Boolean formula conversion.

By experimenting with different pipelines we aim to test the general effects of our compaction approach on probabilistic inference. The 6 pipelines we use for our experiments are listed in Table 3.2. In Chapter 2 these pipelines are indexed as $P4$, $P0$, $P2$, $P13$, $P9$ and $P11$ for ProbLog2/ROBDD, ProbLog2/sd-DNNF (c2d), ProbLog2/sd-DNNF (DSHARP), MetaProbLog/ROBDD, MetaProbLog/sd-DNNF (c2d) and MetaProbLog/sd-DNNF (DSHARP) respectively.

The pipeline implementations of ProbLog not only allow one transformation step to be substituted by another, but can easily be extended with additional transformation steps, such as the compaction method. We employ our detection/compaction algorithm (i) before and (ii) after the cycle handling

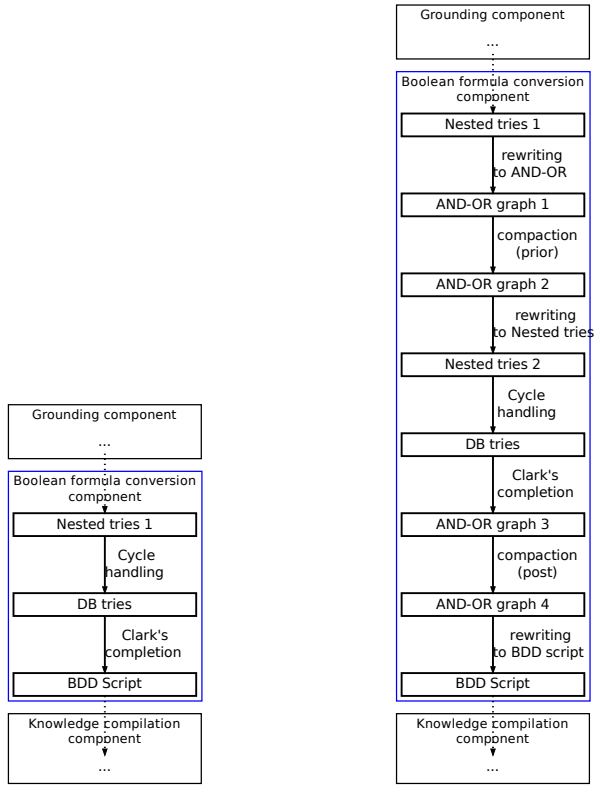
processing of the Boolean formula in any ProbLog pipeline. In (i), called the *prior* compaction, the output of the grounding is represented as an AND-OR graph and then processed by our algorithm. In ProbLog2 the cycle-handling mechanism applies directly on the AND-OR graph and generates a cycle-free AND-OR graph. In MetaProbLog the nested trie structure generated by the Grounding can be rewritten as an AND-OR graph to allow *prior* compaction and then converted back to nested tries. The cycle-handling in MetaProbLog operates on the nested trie structure and produces a BDD Script which is easily rewritten as an AND-OR graph. This allows (ii), that is, to invoke the compaction algorithm again and attempt a further optimization of the (cycle-free) AND-OR graph before the knowledge compilation step. We call this the *post* compaction. Furthermore, we can invoke the *prior* and *post* compactions in the same pipeline; we refer to this compaction setting as *both*.

In Figure 3.1 and Figure 3.2 we show how ProbLog pipelines are transformed in order to incorporate the AND-OR graph compaction algorithm, and in particular the *both* compaction setting which includes *prior* and *post*, for the default MetaProbLog and ProbLog2 pipelines accordingly. In Table 3.2 these pipelines are listed as MetaProbLog/BDD and ProbLog2/sd-DNNF (Dsharp) accordingly. The other 4 pipelines are generated by either changing the knowledge compilation method or interchanging the grounding and the corresponding representation of the grounding output (not shown in the figures).

We see from Figure 3.2 that incorporating AND-OR graph compaction in a ProbLog2-based pipeline does not require any additional preprocessing steps before or after the cycle handling. Compaction is performed on the AND-OR graph generated from the relevant ground LP in the case of *prior* compaction or on the AND-OR graph that results from cycle handling. This is possible because ProbLog2 uses an AND-OR graph representation of the relevant ground LP by default. While incorporating our compaction algorithm in a MetaProbLog-based pipeline requires to augment the pipeline with additional preprocessing in order to generate an AND-OR graph (both for the *prior* and for the *post* compactions) on which to apply the compaction, as shown in Figure 3.1. Such transformations are not computationally expensive, i.e., in linear time to the size of the nested tries or the DB tries we can generate an AND-OR graph and do not affect the overall system performance.

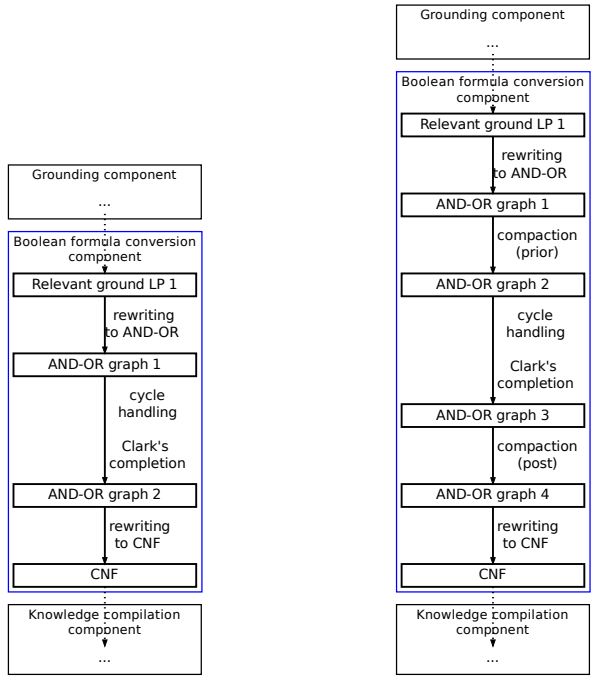
3.4.2 Experimental Set-Up

We experiment on 7 benchmark sets with between 1 and 6 instances each and a total of 709 programs, summarized in Table 3.3. These benchmark are presented in detail in Chapter 2. Moreover, they are the same as the ones used



a. Pipeline without compaction. b. Pipeline with compaction.

Figure 3.1: Incorporating the AND-OR graph compaction algorithm in the default MetaProbLog pipeline for the *both* (that is, *prior* and *post*) compaction setting.



a. Pipeline without compaction. b. Pipeline with compaction.

Figure 3.2: Incorporating the AND-OR graph compaction algorithm in the default ProbLog2 pipeline for the *both* (that is, *prior* and *post*) compaction setting.

in the experiments in Chapter 2[5]. The variety of these benchmarks and the different inference tasks ensure a realistic estimate of the gain or the loss in the performance of ProbLog pipelines due to our compaction algorithm.

Each program is executed with the 6 ProbLog pipelines and the 4 compaction settings – *none*, *prior*, *post* and *both*. Their combination results in 24 different ProbLog pipelines to run each benchmark program with. We chose a timeout of 540 seconds for each test run. We managed to solve 616 out of the 709 programs within the 540 second timeout with at least one of the 24 pipelines.

In order to present our data in a more comprehensive way we divide our benchmarks for which at least one pipeline with one compaction setting succeeds, in three groups: 387 *easy* programs which consume less than 10 seconds; 99

[5]In this chapter, we use more programs from the "Balls" benchmark set and more instances for the "Smokers" and "WebKB" benchmarks.

Name:	Generated from:	Number of benchmark instances:	Number of programs in one instance:	Total number of programs:	Cyclic:	Inference task:
1. Alzheimer	Real-world data	6	17	102	Yes	MARG
2. Balls	Artificial data	1	120	120	No	MARG
3. Dictionary	Real-world data	1	100	100	Yes	MARG
4. Grid	Artificial data	1	15	15	No	MARG
5. Les Miserables	Real-world data	1	60	60	Yes	MARG
6. Smokers	Artificial data	5	24	120	Yes	COND
7. WebKB	Real-world data	4	48	192	Yes	COND

Table 3.3: Summary of the benchmarks used in our experiments.

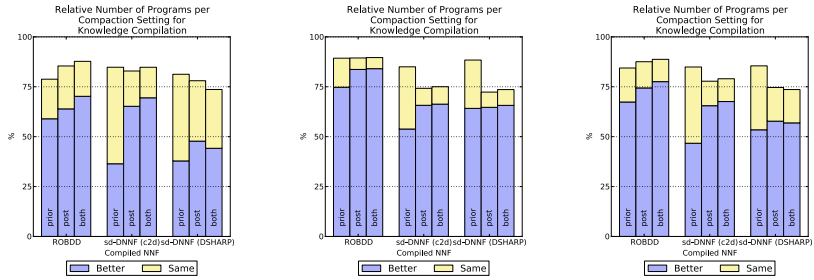
medium programs which consume between 10 and 60 seconds and 150 *hard* programs which consume more than 60 seconds. To classify a program we use the total runtime for the MetaProbLog/ROBDD pipeline without compaction – the default MetaProbLog pipeline.

The sd-DNNF compilers are non-deterministic [8, 54], meaning that for the same CNF the compiled sd-DNNFs may differ. That is why we run each test invoking c2d or DSHARP 5 times (8 pipelines use c2d and 8 use DSHARP). Then we report the average time consumed by the test. From previous experiments we have determined this number to give a reliable estimate of the performance of sd-DNNF compiler.

3.4.3 Experimental Results

In our experiments we collect the total run time for executing a benchmark program. We use the time results to determine the compaction setting which leads to (i) the lowest run times for knowledge compilation; (ii) the lowest total run times; and (iii) the lowest number of timeouts for each of the pipelines and each benchmark set.

Run times for knowledge compilation: We present the number of benchmark programs for which knowledge compilation performs better when compaction ({*prior*, *post*, *both*}) is enabled then when no compaction is used in Figure 3.3. We use the run times for knowledge compilation to determine the best performing knowledge compilation method. We present the best performing knowledge compilation method for MetaProbLog-based pipelines (namely, MetaProbLog/ROBDD, MetaProbLog/sd-DNNF (c2d) and MetaProbLog/sd-DNNF (DSHARP)) and for ProbLog2-based pipelines (namely, ProbLog2/ROBDD, ProbLog2/sd-DNNF (c2d) and ProbLog2/sd-DNNF (DSHARP)) separately in Figure 3.3.a and Figure 3.3.b respectively. We also show the average over the MetaProbLog and ProbLog2-based pipelines in Figure 3.3.c. The figure is created according to the following measurement: for



a. MetaProbLog-based pipelines

b. ProbLog2-based pipelines

c. Average for all pipelines

Figure 3.3: Relative number of programs for which knowledge compilation performs best with the different compaction settings.

each ProbLog pipeline and each compaction setting $c \in \{prior, post, both\}$ we count (i) the number of programs for which knowledge compilation performs better compared to no compaction setting ($c = none$); (ii) the number of programs for which the performance of the knowledge compilation is the same – within a 5% insignificance interval; and (iii) the number of programs for which knowledge compilation when $c = none$ performs better. We measure the run time for knowledge compilation as part of a ProbLog inference pipeline. The 540 seconds timeout is a limit for the total run time rather than for the separate components of the pipeline. Because timeout can be caused due to high run time of component prior to knowledge compilation our measurements include only programs for which neither the inference with compaction nor with no compaction times out. We give these numbers as percentages. For clarity we show only (i) and (ii); (iii) is the remaining up to 100%.

From Figure 3.3 we conclude that compaction improves between 35% (see Figure 3.3.a sd-DNNF (c2d), *prior* compaction) and 80% (see Figure 3.3.b ROBDD, *both* compaction) of the executed benchmark programs, while it increases the run time of the knowledge compilation method in 15% (see Figure 3.3.b ROBDD, *both* compaction) to 25% (see Figure 3.3.b sd-DNNF (DSHARP), *post* compaction) of the cases. Its effect is more salient for knowledge compilation to ROBDDs than to sd-DNNs. While on average (see Figure 3.3.c) our algorithm improves the knowledge compilation time for the majority of the benchmarks, between 73% to 80% for ROBDDs, between 48% to 70% for sd-DNNFs compiled with c2d and between 52% to 55% for DSHARP. One reason for the less positive effect on knowledge compilation to sd-DNNF is that although our compaction method decreases the size of the AND-OR graph, it may introduce such subgraphs with a corresponding CNF that is difficult for

DSHARP to compile or the compiled sd-DNNF is difficult to evaluate. Moreover, it is often the case that redundant information can improve the heuristics used by the sd-DNNF compilers[6] while our algorithm removes any such information aiming at the most compact AND-OR graph. It is worth noting that while DSHARP suffers from this problem, c2d is not influenced in that extent. The two compilers differ in their implementation and while DSHARP aims fast sd-DNNF compilation, c2d applies more intelligent techniques to compile a CNF into sd-DNNF, including optimizations which bypass the aforementioned issue.

Best performing compaction setting (total run time): For each ProbLog pipeline and each compaction setting $c \in \{prior, post, both\}$ we count (i) the number of programs that perform better compared to no compaction setting ($c = none$); (ii) the number of programs for which the performance is the same – within a 5% insignificance interval; and (iii) the number of programs for which $c = none$ performs better. We exclude programs for which both the inference with compaction and with no compaction times out. We give these numbers as percentages in Figure. 3.4. For clarity we show only (i) and (ii); (iii) is the remaining up to 100%.

In comparison to Figure 3.3 in Figure 3.4 we include the time for compaction, AND-OR graph generation (in case of grounding to nested tries) and Boolean formula conversion. We also divide the set of programs in three subsets, making it easier to evaluate the effect of our method. Furthermore, in contrast to Figure 3.3 which compares the run times for programs that were executed successfully within the timeout limit both with compaction and with no compaction, Figure 3.4 compares programs for which one pipeline with at least one compaction setting (including no compaction) terminates within the timeout limit. That is, in Figure 3.4 some programs may timeout and still be measured.

Figure 3.4 does not quantify the gain or the loss due to compaction but only presents percentile wise how many programs gain from compaction. We present the gain due to compaction in Figure 3.5. For each ProbLog pipeline and each compaction $c \in \{prior, post, both\}$ we sum (a) the gain in the total run time for each benchmark compared with the runtime of no compaction ($c = none$) when the compaction performs better: T_C^g; (b) the loss in the total run time due to compaction, that is, when no compaction performs better: T_N^g. We normalize both gain and loss by dividing by the total number of programs within a benchmark set to compensate for the fact that some of them contain more programs. We exclude programs for which both inference with compaction and with no compaction times out. We show the gain due to compaction relative to

[6]For example, [39] identifies the backbone [53] of a CNF formula and conjoin it with the original formula aiming to speed-up knowledge compilation.

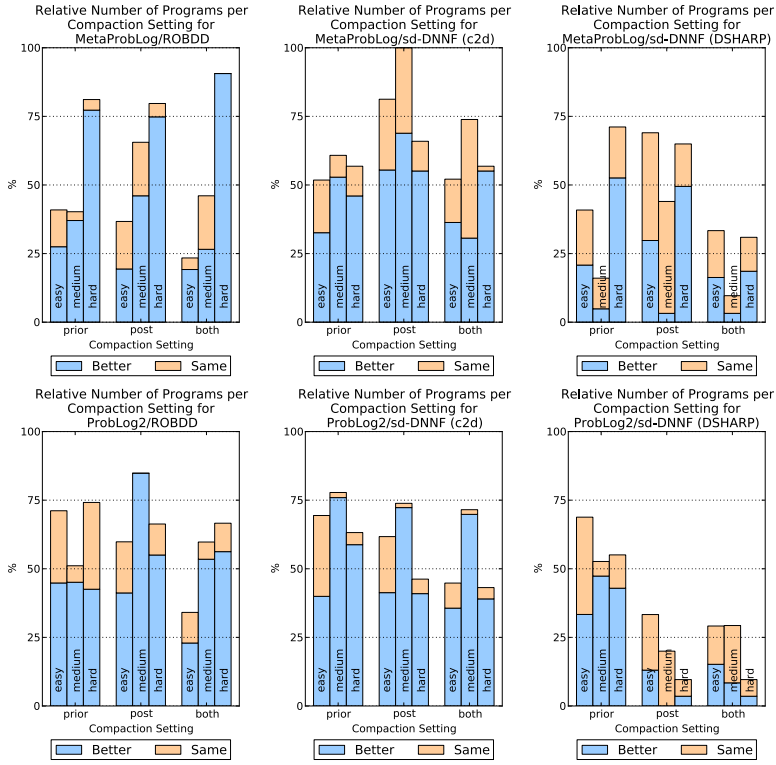


Figure 3.4: Relative number of programs with best total runtime for the different compaction settings.

the sum of gain and loss: $\frac{T_C^g}{T_C^g+T_N^g}$ in Figure 3.5. These results illustrate the run time gain due to compaction in percentage. The 50% are marked with a bold dashed line; values above that line state that compaction is better.

For the MetaProbLog/ROBDD pipeline compaction improves the inference for the medium and hard problems but not that much for the easy problems (see Figure 3.4 and Figure 3.5). This pipeline is in general very fast (see Chapter 2) and the time spend to perform our algorithm for the easy problems is not compensated from the gain in knowledge compilation. For MetaProbLog/sd-DNNF (c2d) and ProbLog2/ROBDD using compaction improves the performance on all problems and mostly for the medium problems. The average highest gain due to compaction appears for the ProbLog2/ROBDD pipeline and is between 75% and 98% for all compaction settings and problem sizes.

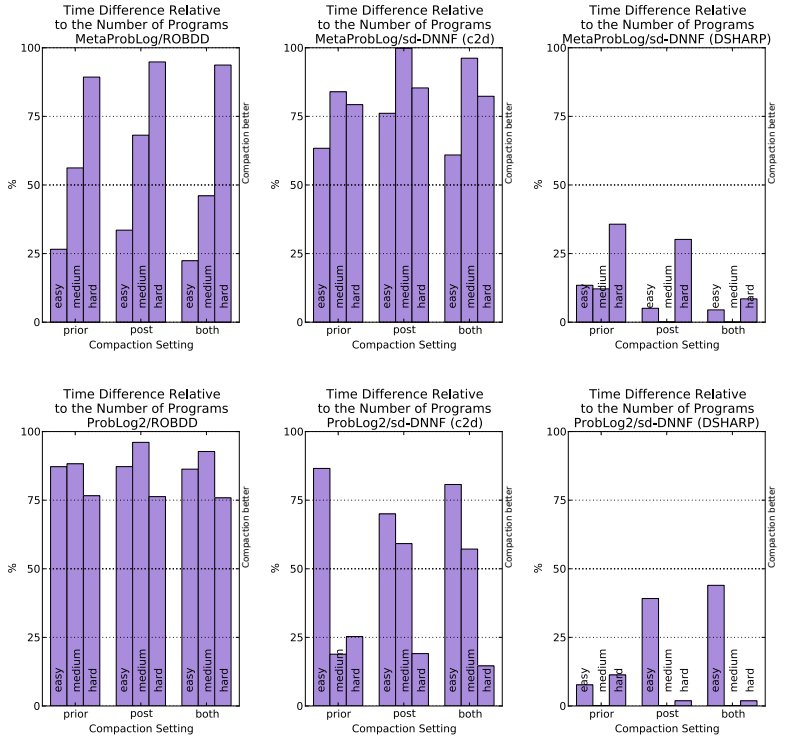


Figure 3.5: Relative time gain due to a specific compaction.

The results summarized in both Figure 3.4 and Figure 3.5 show that compaction affects negatively pipelines that use the DSHARP sd-DNNF compiler. In particular for the case of DSHARP compaction improves inference for 5% to 50% of the benchmarks (Figure 3.4) and leads to run time gain of 0% to 43% (Figure 3.5). The reasons for the worse performance in the case of the DSHARP compiler relate to the CNF of the compacted Boolean formula. We discuss in detail this issue in Section 3.4.4.

We can conclude that in general for all but the pipelines with knowledge compilation to sd-DNNF with DSHARP using compaction is preferable to no compaction, that is for the MetaProbLog/ROBDD, MetaProbLog/sd-DNNF (c2d), ProbLog2/ROBDD pipelines and ProbLog2/sd-DNNF (c2d).

Also we observe that none of the compaction settings (i.e., the *prior*, *post* or *both*) outperforms the other compaction settings. Comparing the results in Figure 3.4 and Figure 3.5 we see that for MetaProbLog/ROBDD the *both* compaction is

preferable; for MetaProbLog/sd-DNNF (c2d) and ProbLog2/ROBDD pipelines preferable is the *post* compaction; for ProbLog2/sd-DNNF (c2d) the *prior* compaction. The *post* and *both* compactions often yield the same Boolean formulae. In such cases *both* spends unnecessary extra time for *prior* compaction.

Least timeouts compaction setting: For each benchmark set d we count the total number of programs solved within the 540 seconds by at least one pipeline and compaction setting (we denote this number as S_d); for each pipeline p and compaction setting c we count the number of programs from the benchmark set d which time out – $T_{d,p,c}$. We ignore the programs for which all pipelines time out. We also compute the total number of programs which time out for one pipeline and one compaction setting: $\sum_{d \in D} T_{d,p,c}$; and we compute the accumulated ratio $\sum_{d \in D} \frac{T_{d,p,c}}{S_d}$, where D is the set of all benchmark sets. For the "Les Miserables", "Smokers 1", "Smokers 2" and "Smokers 4" sets there is no one pipeline that succeeds over all 636 programs we managed to solve. The lowest number of timeouts for these cases are larger than 0. For these programs at least one of the other pipelines does succeed within the 540 seconds. These results are shown in Table 3.4.

Using the timeout results we notice that compaction allowed us to solve significantly more problems that would otherwise timeout (see Table 3.4). Particularly, in the best case, MetaProbLog/ROBDD with *both* compaction, we can solve 38% more programs; ProbLog2/ROBDD with *post* compaction can solve 37% more programs. The two pipelines which use compilation to sd-DNNF with c2d benefit less from compaction as noted by the other results, MetaProbLog/sd-DNNF (c2d) with *post* compaction can solve 6% more programs while for ProbLog2/sd-DNNF (c2d) compaction introduces up to 12% more timeouts. For the pipelines with knowledge compilation to sd-DNNF with Dsharp compaction introduces extra timeouts: MetaProbLog/sd-DNNF (Dsharp) – 5%, 6% and 7% for *prior*, *post* and *both* compaction settings; ProbLog2/sd-DNNF (Dsharp) – 23%, 20%, 20% for *prior*, *post* and *both* compaction settings. The extra timeouts occur for the "WebKB", the "Smokers" and the "Balls" benchmarks. The reason is that the "WebKB" and the "Smokers" benchmark sets contain multiple queries and evidence; in the "Balls" benchmarks our method detects and compacts a lot of branch patterns (Pattern 3) which affect negatively the Dsharp compiler (see Section 3.4.4). Often the query and evidence atoms appear also as subgoals. Queries and evidence are required for the final step of WMC thus they should not to be removed from the Boolean formula. Therefore there are less patterns that can be compacted in the case of COND with respect to MARG tasks. For the other benchmarks compaction reduces the overall amount of timeouts.

		No Compaction						Prior Compaction						Post Compaction						Both Compactions					
Benchmark/Instance	# Programs solved	MetaProbLog/ROBDD	MetaProbLog/sd-DNNF(c2d)	MetaProbLog/sd-DNNF(DSHARP)	ProbLog2/ROBDD	ProbLog2/sd-DNNF(c2d)	ProbLog2/sd-DNNF(DSHARP)	MetaProbLog/ROBDD	MetaProbLog/sd-DNNF(c2d)	MetaProbLog/sd-DNNF(DSHARP)	ProbLog2/ROBDD	ProbLog2/sd-DNNF(c2d)	ProbLog2/sd-DNNF(DSHARP)	MetaProbLog/ROBDD	MetaProbLog/sd-DNNF(c2d)	MetaProbLog/sd-DNNF(DSHARP)	ProbLog2/ROBDD	ProbLog2/sd-DNNF(c2d)	ProbLog2/sd-DNNF(DSHARP)	MetaProbLog/ROBDD	MetaProbLog/sd-DNNF(c2d)	MetaProbLog/sd-DNNF(DSHARP)	ProbLog2/ROBDD	ProbLog2/sd-DNNF(c2d)	ProbLog2/sd-DNNF(DSHARP)
alzheimer/1	5	2	4	4	1	4	4	2	4	4	**0**	4	4	1	4	4	1	4	4	1	4	4	1	4	4
alzheimer/2	9	4	6	8	**0**	5	6	3	6	8	**0**	5	6	3	6	8	**0**	5	6	3	6	8	**0**	6	6
alzheimer/3	9	**0**	**0**	**0**	3	3	3	**0**	**0**	**0**	3	3	3	**0**	**0**	**0**	3	3	3	**0**	**0**	**0**	3	3	3
alzheimer/4	17	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
alzheimer/5	9	3	3	6	1	5	5	3	3	6	3	5	5	1	3	6	3	4	6	**0**	3	6	3	4	6
alzheimer/6	6	1	3	5	**0**	2	4	**0**	3	5	**0**	2	4	**0**	3	5	**0**	2	3	**0**	3	5	**0**	2	3
balls	120	**0**	**0**	2	5	**0**	7	**0**	**0**	72	**0**	**0**	84	**0**	**0**	76	**0**	**0**	75	**0**	**0**	76	**0**	**0**	70
dictionary	100	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
grid	12	4	**0**	7	1	**0**	7	4	2	6	1	1	7	4	1	6	1	1	7	4	1	6	1	1	7
les_miserables	47	**2**	10	27	**2**	27	28	**2**	14	27	**2**	24	28	**2**	16	27	3	26	28	**2**	21	27	**2**	28	28
smokers/1	21	7	14	14	**1**	4	3	7	13	13	**1**	4	3	8	14	14	**1**	4	3	7	13	14	**1**	4	3
smokers/2	21	6	17	15	2	4	4	6	17	15	**1**	4	3	6	17	16	2	4	5	6	17	16	2	4	5
smokers/3	18	3	11	11	**0**	1	1	3	13	12	**0**	1	1	4	12	11	**0**	1	2	3	13	12	**0**	1	2
smokers/4	20	5	13	13	**1**	3	4	3	14	13	**1**	3	4	5	14	13	**1**	4	5	3	15	13	**1**	4	5
smokers/5	17	1	12	9	**0**	**0**	1	2	12	9	**0**	**0**	1	1	12	9	**0**	**0**	1	3	11	9	**0**	**0**	1
webkb/1	48	18	29	14	9	**0**	**0**	10	14	14	3	**0**	2	8	14	14	3	**0**	2	8	14	14	4	3	3
webkb/2	41	16	27	26	16	11	2	16	21	24	15	8	**1**	16	21	27	16	10	2	16	22	28	16	10	2
webkb/3	48	20	41	41	17	**0**	4	7	41	41	1	12	28	1	41	41	1	14	28	**0**	41	41	1	10	28
webkb/4	48	17	29	14	10	**0**	**0**	8	14	14	4	**0**	3	7	14	14	7	**0**	3	9	14	14	7	3	4
Total:	616	109	219	216	69	69	83	76	191	283	35	76	187	67	192	291	42	82	183	65	198	293	42	87	180
Average timeouts:		4.35	8.68	9.61	2.15	4.01	5.04	3.41	8.27	10.07	1.47	4.20	6.21	2.94	8.22	10.21	1.83	4.28	6.31	2.77	8.35	10.29	1.83	4.47	6.31

Table 3.4: Number of timeouts. In bold we mark the lowest number of timeouts per benchmark set; with underlined we mark the lowest sum and lowest accumulated ratio of the timeouts for a pipeline and a compaction setting.

$$A \Leftrightarrow \bigwedge_{i=1}^{n} B_i \quad \longleftrightarrow \quad \begin{array}{l} A \vee \neg B_1 \vee .. \vee \neg B_n \wedge \\ \neg A \vee B_1 \wedge \\ .. \\ \neg A \vee B_n \end{array}$$

a. Conjunction to CNF.

$$A \Leftrightarrow \bigvee_{i=1}^{n} B_i \quad \longleftrightarrow \quad \begin{array}{l} \neg A \vee B_1 \vee .. \vee B_n \wedge \\ A \vee \neg B_1 \wedge \\ .. \\ A \vee \neg B_n \end{array}$$

b. Disjunction to CNF.

Figure 3.6: Generating a CNF from Boolean subformula containing equivalence.

To summarize, our results show that in general using compaction is beneficial. Although we can indicate the *post* compaction as having the most positive effect on average, it is obvious that the best compaction setting depends on the pipeline, the ProbLog program and the task to be solved. Moreover, we conclude that there is not one best performing pipeline and compaction setting over all benchmarks. On average, the pipeline with the least timeouts was ProbLog2/ROBDD with *post* compaction.

3.4.4 Limitations of the Approach

For knowledge compilation to sd-DNNF the compacted AND-OR graph needs to be translated into a Boolean formula in CNF. There are some complications that relate to this translation. Another issue relates to the number of edges of the compacted AND-OR graph as compared to the unprocessed graph: our patterns ensure a decrease of the number of nodes of the AND-OR graph (see Table 3.1) but they may, in some cases, introduce more edges. In this section we discuss these limitations and how to overcome them.

Translation between AND-OR graphs and Boolean formulae in CNF: Each edge in an AND-OR graph states the dependency between a parent and a child node. I.e., an OR node A with children B_1 to B_n corresponds to $A \Leftrightarrow \bigvee_{i=1}^{n} B_i$[7]; an AND node A with children B_1 to B_n corresponds to $A \Leftrightarrow \bigwedge_{i=1}^{n} B_i$

Such subformulae are easy to rewrite in CNF as shown in Figure 3.6

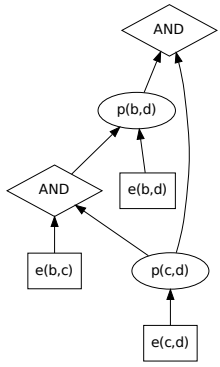
[7]If the AND-OR graph represents a ground logic program correctness of the equivalence is achieved after cycles are handled accordingly.

Exploiting this form of the CNF subformulae allows the translation between an AND-OR graph and a Boolean formula and vice versa without increasing the size of either of them. If a CNF does not contain subformulae that belong to the aforementioned type the translation of a CNF to AND-OR graph is as follows: for each disjunction (i) a new OR node is created in the AND-OR graph and the disjuncts are added as child nodes; and (ii) an AND node is created in the AND-OR graph and all OR nodes associated with a disjunction are added as child nodes to the AND node. This transformation is similar to the translation of CNF to BDD definitions, discussed in Chapter 2. It may increase the size of the AND-OR graph substantially. If our algorithm does not detect enough clusters to compact then the translation of the graph back to CNF may result in a larger CNF than the initial one. This issue is illustrated in Example 3.3.

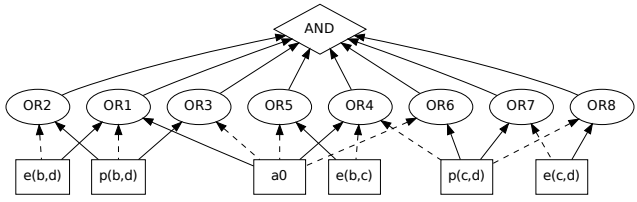
Example 3.3. *Recall the CNF from Example 2.13 in Chapter 2:*

CNF: $(\neg p(b,d) \vee e(b,d) \vee a0) \wedge (p(b,d) \vee \neg e(b,d)) \wedge (p(b,d) \vee \neg a0) \wedge$
$(a0 \vee \neg e(b,c) \vee \neg p(c,d)) \wedge (\neg a0 \vee e_{bc}) \wedge (\neg a0 \vee p_{cd}) \wedge$
$(p_{cd} \vee \neg e_{cd}) \wedge (\neg p_{cd} \vee e_{cd})$

using the transformations in Figure 3.6 we can build the following AND-OR graph:



If we decide to ignore the structure of the CNF and not use the transformations in Figure 3.6 then the corresponding AND-OR graph is:

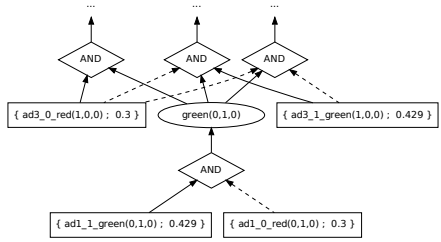


(dashed edges denote negation). △

Rule-based conversion: Chapter 2 presents two approaches for Boolean formula conversion. Namely, the proof-based approach which is used in our compaction experiments and the rule-based. The rule-based approach generates CNF formulae which (i) do not always comply with the aforementioned subformulae type and (ii) contain a lot of negation. Given (i), the AND-OR graph for *post* or *both* compaction may be substantially large. Our compaction doesn't handle AND or OR clusters over negated literals – they are excluded from the literals which form the cluster. Given (ii), our compaction then cannot ensure that all possible AND or OR clusters are compacted. That is why we do not employ compaction for pipelines using the rule-based conversion[8].

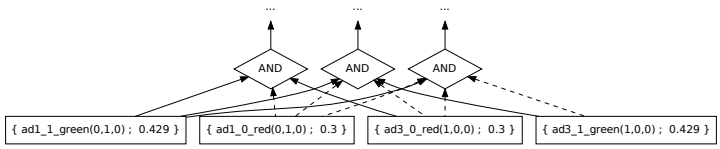
The effect of Branch I patterns compaction on compilation with Dsharp: Branch I pattern involves a node A depending on an OR node B such that B depends only on an AND node C. That is, B is an intermediate node connecting A and C. Removing B and connecting directly C to A is how this pattern is compacted. If A is an AND node then C is also removed and the children nodes of C are connected to A. Assume that there are multiple AND nodes A_1 to A_n that are parents to the node B. The compaction of this pattern will contain edges to connect each node A_i, for $i \in \{1, .., n\}$, with the children of C. Let C has m children, then the graph will contain two nodes less, namely B and C and $(n-1) \times m - 1$ edges more. We illustrate this in Example 3.4.

Example 3.4. *A fragment of the AND-OR graph associated with the relevant ground program of benchmark from the "Balls" set contains a Branch I pattern:*



Applying the compaction transformation results in:

[8]We implemented and ran preliminary tests on pipelines with the rule-based Boolean formula conversion. They show that compaction is inefficient for the majority of the cases.



We see that now the number of incoming connections to each AND node has increased. △

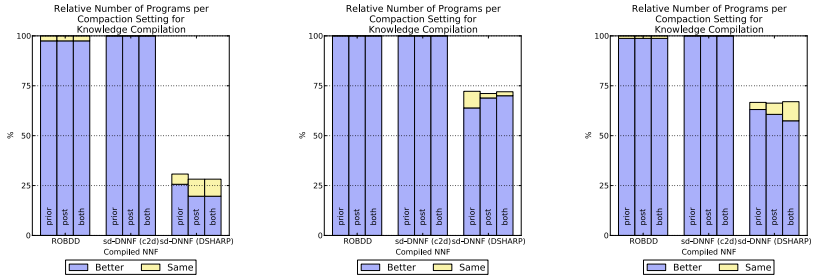
CNFs generated from such AND-OR graphs contain more clauses and some clauses may contain more literals, compared to a CNF generated from an AND-OR graph without the Branch I pattern compaction; a BDD script generated from such a graph will contain less BDD definitions, some of which may have more literals in their bodies and we do not notice negative effects on knowledge compilation to ROBDDs. The search approach employed by the Dsharp compiler often can benefit from additional information in the CNF, e.g., constraints. Since applying compaction for Branch I patterns removes such information, pipelines that use knowledge compilation with Dsharp and employ compaction do not perform well for problems that produce such patterns.

Although removing some clauses by applying Branch I pattern compaction decreases the performance of Dsharp it does not have such an effect on the c2d compiler. c2d implements different optimizations aiming at more efficient sd-DNNFs while Dsharp aims at efficient knowledge compilation but the resulting sd-DNNFs cannot be evaluated efficiently.

We detected Branch I patterns in 5 out of the 7 benchmark sets running ProbLog2-based pipelines and in all 7 benchmark sets for MetaProbLog-based pipelines (see Appendix C). Figure 3.7 shows the number of programs for which knowledge compilation performs better when compaction ($\{prior, post, both\}$) is enabled than when no compaction is used for the "Balls" benchmark set. This benchmark set includes a lot of Branch I patterns. The results are drawn according to the measurement presented in Figure 3.4.

To show that compaction without the Branch I pattern improves Dsharp compilation we experimented on the "Balls" benchmark set after disabling detecting and compacting this pattern. The results are presented in Figure 3.8.

Figure 3.8 shows that using compaction without Branch I pattern improves inference for up to 99% of the programs.

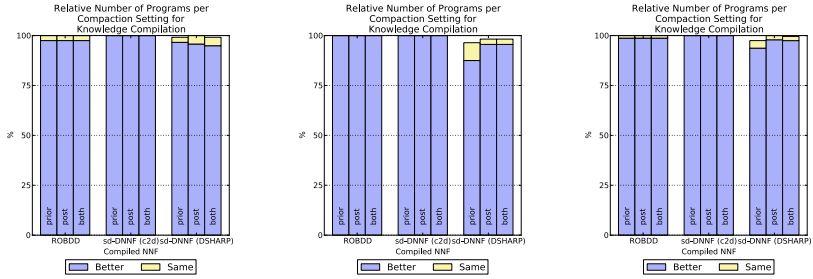


a. MetaProbLog-based pipelines

b. ProbLog2-based pipelines

c. Average for all pipelines

Figure 3.7: Relative number of programs for which knowledge compilation performs best with the different compaction settings for the "Balls" benchmark set. Detecting and compacting Branch I pattern is enabled.



a. MetaProbLog-based pipelines

b. ProbLog2-based pipelines

c. Average for all pipelines

Figure 3.8: Relative number of programs for which knowledge compilation performs best with the different compaction settings for the "Balls" benchmark set. Detecting and compacting Branch I pattern is **disabled**.

3.5 Conclusion and Future Work

This chapter presented a pattern-based approach for compacting Boolean formulae. It detects and compacts 6 (out of 7 identified) patterns – 4 that preserve logic equivalence and 2 that preserve equivalence with respect to the weighted model count of the Boolean formula. Our approach aims to improve probabilistic inference that uses knowledge compilation and weighted model counting. It targets but is not limited to the probabilistic logic ProbLog and its underlying implementations.

We performed experiments with 6 different ProbLog pipelines and 3 compaction settings on 7 benchmark sets with 709 benchmarks in total. Our results show that compaction improves knowledge compilation to ROBDDs with the compiler SimpleCUDD as well as to sd-DNNFs with the compiler c2d; our approach increases the run time for knowledge compilation to sd-DNNFs with the compiler Dsharp. We identified that this decrease in the performance of Dsharp relates to the translation of compacted AND-OR graphs as CNF. The gain in the total runtime due to compaction is most salient for harder problems. The decreased amount of timeouts proofs that our approach enables inference on problems unsolved before (i.e., without compaction).

In the future we want to investigate non-compacting transformations that could aid (thus improve) the knowledge compilation. We already showed that knowledge compilation by Dsharp can be improved by ignoring one of our patterns. In addition, we plan to extend our algorithm to handle problems outside the domain of ProbLog. We aim to test it on benchmarks from [39] in order to determine its general effects.

Part II

The Extended ProbLog Language

Outline Part II

ProbLog is a probabilistic logic and learning framework – a language and a system. The *core ProbLog language* supports probabilistic facts (facts whose truth values are determined probabilistically), non-probabilistic facts (as in Prolog), rules which define deterministic consequences of the facts (probabilistic or non-probabilistic) and built-ins. The core ProbLog language is expressive enough to encode a wide range of problems [31, 76, 57]. But it lacks expressivity for others and the user is required to write complicated ProbLog programs in order to solve them. In this part we present the *extended ProbLog language* that supports *constraints* and *annotated disjunctions*.

In [15] the authors propose to extend the expressive power of ProbLog by introducing new language constructs, namely *constraints* – First-Order Logic sentences that have to hold. They introduce the intended semantics of constraints for ProbLog and argue about the reasons such an extension would improve the expressivity of the ProbLog language. They do not, though, provide an implementation. We built upon this idea to define the syntax of this extension, called cProbLog, and then implement a method for probabilistic inference with constraints. In **Chapter 4** we discuss the semantics of cProbLog, introduce its syntax and present our inference approach. We also discuss alternatives and present an optimization method for reducing the size of the grounding. We then give a series of examples in order to familiarize the user with the new language constructs.

In the second chapter of this part, **Chapter 5**, we discuss another extension to the core ProbLog language, namely the annotated disjunctions. Annotated disjunctions [87, 50, 86] are basic probabilistic constructs suitable to encode mutually exclusive random events. In [21, Chapter 3], Gutmann introduces an approach to convert annotated disjunctions into core ProbLog language. This conversion is correct for the MARG and COND tasks but not for the MPE task. We developed a method based on constraints to correctly reason with annotated disjunctions regardless the inference task. Our approach transforms

an annotated disjunction to a set of probabilistic facts and rules and uses constraints to restrict the possible worlds to the correct ones with respect to the initial annotated disjunction.

Chapter 4

cProbLog: ProbLog with FOL Constraints

In [15] the authors introduce the notion and intended semantics of constraints for ProbLog. Constraints are First-Order Logic (FOL) sentences that need to hold. The work of Fierens et al. does neither define the syntax nor provide any implementation of constraints for ProbLog. We build upon these ideas and define the syntax of cProbLog – the extension of ProbLog with constraints; then we built the first implementation of an inference method for ProbLog programs with constraints.

For a given ProbLog program, constraints make some possible worlds invalid and the probability of a query needs to be computed with respect to the remaining possible worlds, in which the constraints hold. Evidence states which atomic choices are true or false and also restrict the set of possible worlds to the ones where the atomic choices hold. We present constraints as generalization of evidence to FOL sentences and build our method for inference with constraints according to this property. Our inference method, called the *constraint-evidence* approach, translates constraints into the core ProbLog language. More precisely, constraints are translated into rules and evidence is imposed that validates the same possible worlds as the constraints. We implement our constraint-based approach into ProbLog2 and evaluate its performance. Our design allows not only to incorporate the support for cProbLog constraints in other ProbLog pipelines but also to extend other probabilistic logic formalisms; we implemented an extension of CLP($\mathcal{BN}$) [64] with cProbLog constraints.

In order to improve the inference performance we build an optimization method

for constraints that reduces the grounding size. This method removes parts of the AND-OR graph that do not contribute to the probability calculation. We show the effectiveness of this optimization on a number of benchmarks.

We present a series of examples of cProbLog programs from various domains. These examples aim to familiarize the user with this extension of the ProbLog language.

We also compare cProbLog to other probabilistic logic formalisms that use constraints. In particular, these are PCLP [51], CLP($\mathcal{BN}$) [64] and CHRiSM [77].

The main contribution of our work is the design and the implementation of the constraint-evidence approach that adds support of FOL constraints for ProbLog. The modular design of the ProbLog inference pipeline allows us to implement our method as a stand-alone tool. This design facilitates the incorporation of such a tool in other systems as we did in CLP($\mathcal{BN}$) [64]. We also introduced an optimization method to reduce the size of the grounding of constraints that improves the inference performance.

4.1 Motivation

In Chapter 2 we presented the syntax and the semantics of ProbLog. We called the language that supports probabilistic facts that encode uncertain events, non-probabilistic (Prolog) facts, rules to determine logic consequences of the probabilistic and non-probabilistic facts and built-ins, the *core* ProbLog language. An example ProbLog program is given in Example 4.1.

Example 4.1 (Road map)**.** *Consider four cities* `c1`*,* `c2`*,* `c3` *and* `c4`*. There exist roads that connect directly city* `c1` *with city* `c2`*, city* `c2` *with city* `c4`*, city* `c1` *with city* `c3` *and city* `c3` *with city* `c4`*. Moreover, each road between city* $\texttt{c}_i$ *and* $\texttt{c}_j$ *is characterized by the uncertainty to reach* $\texttt{c}_j$ *starting* $\texttt{c}_i$ *on time. The following ProbLog program encodes this small road map together with the uncertainties:*

```
0.7::road(c1, c2).      0.7::road(c2, c4).
0.5::road(c1, c3).      0.9::road(c3, c4).

reach(A, B):- road(A, B).
reach(A, B):- road(A, A1), reach(A1, B).
```

Each probabilistic fact $\texttt{p}_{ij}$`::road(`$\texttt{c}_i$`, `$\texttt{c}_j$`).` *states that the road connecting cities* $\texttt{c}_i$ *and* $\texttt{c}_j$ *allows one to reach* $\texttt{c}_j$ *on time with probability* $\texttt{p}_{ij}$*.* △

Recall from Chapter 2 that each probabilistic fact in a ProbLog program is either true or false in a different *possible world*. ProbLog defines a distribution over all possible worlds of a ProbLog program. A query atom is true in a subset of these possible worlds. E.g., for the ProbLog program in Example 4.1 the query `reach(c1, c4)` is true in 7 out of the 16 possible worlds. Each possible world has a probability – it is the product of the probabilities of all probabilistic facts that are true and (1-the probabilities) of all probabilistic facts that are false in that possible world. The marginal (MARG) probability of a query is the sum of the possible worlds in which the query is true. For the ProbLog program in Example 4.1, the MARG probability of the query `reach(c1, c4)` is $P(\texttt{reach(c1, c4)}) = 0.7195$.

In ProbLog, observations about the truth values of some atoms of a ProbLog program are encoded as evidence. Evidence is a set of pairs (a_i, α_i) with α_i the observed truth value of the ground atom a_i ($\alpha_i \in \{true, false\}$). We denote with $E = \{a_1, .., a_n\}$ the set of evidence atoms and with $e = \{\alpha_1, .., \alpha_n\}$ – their corresponding truth values. Evidence restricts the set of possible worlds of a ProbLog program to the ones in which each $a_i = \alpha_i$ is valid. A query q can be true in some of these possible worlds. These are the ones in which the conjunction $q \wedge E = e$ is true: $\Omega^{q \wedge E=e} \subseteq \Omega^q \subseteq \Omega$ (for Ω the set of possible worlds of a given ProbLog program). Using Bayes' rule ProbLog computes the conditional (COND) probability $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$. For instance, for the ProbLog program in Example 4.1 the observation "*the road between city 3 and city 4 is not available*" can be given as the evidence `road(c3, c4)`$= false$. Then all possible worlds where the atom `road(c3, c4)` is true will become invalid and the COND probability of a query will be calculated with respect to the other worlds (in which `road(c3, c4)` is false).

In ProbLog, evidence can express observations on single atoms[1]. That is why encoding more complex additional knowledge can become cumbersome as shown in Example 4.2.

Example 4.2. *In order to state that the road between city* `c2` *and city* `c4` *or the road between city* `c3` *and city* `c4` *is available, that is, it is certain that you will reach city* `c4` *from city* `c2` *or city* `c4` *from city* `c3` *on time, it is required to modify the initial ProbLog program of Example 4.1 into:*

```
0.7::road(c1, c2).      0.7::road(c2, c4).
0.5::road(c1, c3).      0.9::road(c3, c4).
```

[1] In the mainstream ProbLog2 implementation [14] evidence is allowed only on ground atoms. The current release ProbLog2.1, allows also evidence on non ground atoms. Evidence on a non ground atom corresponds to evidence on all ground atoms that can be derived by instantiating the variables.

```
reach(A, B):- road(A, B).
reach(A, B):- road(A, A1), reach(A1, B).

%ADDITIONAL CODE:
add_ev:- road(c2, c4).
add_ev:- road(c3, c4).
```

and state the evidence that the atom `add_ev` *is true.* △

Often First-Order Logic (FOL) sentences are more expressive than a Prolog program. For example, when encoding logic statements that interleave universally and existentially quantified variables, or mutually exclusive statements using the logic operation of XOR, etc.

In [15], the authors introduce an extension of the ProbLog language with FOL sentences to represent additional knowledge to the ProbLog program, called cProbLog. These FOL sentences encode *constraints* that need to hold. Similar to evidence, a constraint holds for a certain subset of the possible worlds of a ProbLog program. So, the probability of a query can be computed with respect to the possible worlds which are valid according to the constraints. That is why we see constraints as generalization of evidence to FOL sentences.

Example 4.3. *We can use a single FOL sentence to encode the additional knowledge (or condition) in Example 4.2:*

$$\texttt{road(c2, c4)} \vee \texttt{road(c3, c4))}.$$

but also a more complex logic formula like:

$$\forall X\ \texttt{city}(X) \wedge \texttt{road}(X\texttt{, c4}) \wedge X \neq \texttt{c4}$$

△

Example 4.3 shows two constraints – one ground and another that contains a universally quantified variable. The second constraint can be written as a Prolog rule with all possible ground atoms derived from instantiating the variable X in the body, e.g., `add_ev:- reach(c1,c4), reach(c2,c4), reach(c3,c4).`, assuming X can have the values `c1`, .., `c4`. This formulation, though, is impractical for more complicated logic formulae.

4.2 Syntax and Semantics

4.2.1 Syntax

The extension of the ProbLog language with FOL constraints, called cProbLog consists of (i) facts – probabilistic and non-probabilistic, (ii) logic programming rules, (iii) built-ins and (iv) FOL sentences that encode constraints.

The FOL sentences of cProbLog constraints use standard logic operators: $\forall$, $\exists$, $\wedge$, $\veebar$, $\vee$, $\neg$, $\Leftrightarrow$ and $\Rightarrow$, represented in cProbLog by the operators `for_all`, `exists`, `and`, `or`, `xor`, `not`, `iff` and `implies`. The quantifier delimiter is written as `:` and parentheses are used to express priority.

Example 4.4 (Packing problem)**.** *Consider a problem where a set of items need to be packed in a suitcase. Each item has a certain weight and a probability to be packed. The total weight of the suitcase needs to remain within a predetermined limit.*

```
weight(skis,6).        weight(board, 8).
weight(boots,4).       weight(helmet,3).
0.16::pack(skis).      0.125::pack(board).
0.25::pack(boots).     0.33::pack(helmet).
inlimit(Limit):- inlimit([skis,boots,board,helmet],Limit).
inlimit([],Limit):- Limit>=0.
inlimit([I|R],Limit):- pack(I), weight(I,W),
                        L is Limit-W, inlimit(R,L).
inlimit([I|R],Limit):- \+pack(I), inlimit(R,Limit).
```

This program can be used to query for the probability of packing together certain items that weigh in total less than or equal to 10*, that is, use the query* `inlimit(10)`*. In order to add additional restrictions like "if the skis are packed then also the boots need to be packed" we use the constraint:*

```
pack(skis) implies pack(boots)
```

△

Example 4.4 shows a constraint with two ground probabilistic atoms. The generalization power of the cProbLog language comes with the support of non ground constraints – FOL sentences with quantified variables. In a cProbLog constraint the range of values of every quantified variable need to be explicitly defined. That is in order to keep the grounding of the FOL sentences finite and

also for determining the relevant grounding. ProbLog is an untyped language hence we do not introduce type declarations for cProbLog, but rather an explicit enumeration of values, that is, a *domain*, in the constraint. There are two ways to express a domain: either as a set enumerating all possible values of a specific variable or as a call to a predicate; this predicate serves as a generator for the values of the variable. In principle the domain definition can be a more complex Prolog expression as long as it generates the values for the variables. To link a variable to a certain domain given as a Prolog predicate we use the built-in `of/2`; for a domain specified as a set we use `in/2`.

Example 4.5 (Packing problem continued)**.** *Assume it is required to have at least* 2 *items in the suitcase and one of them should be the boots. We can encode this constraint in two ways:*

1.
```
pack(boots) and exists X of item(X) :
        pack(X) and not X == boots.
```

2.
```
pack(boots) and exists X in {skis, boots, board, helmet} :
        pack(X) and not X == boots.
```

The first formulation requires that the ProbLog program defines a predicate `item/1` *that, naturally, specifies all items that can possibly be packed. It acts as value enumerator. The second one defines the domain of the variable* `X` *as a set.*

We can simplify the constraint by omitting `boots` *from the domain of* `X` *and remove* `not X == boots.`*:*

```
pack(boots) and exists X in {skis, board, helmet} : pack(X)
```

△

ProbLog2 uses the built-in predicates `query/1` and `evidence/2` to declare queries and evidence in a program. Since we implement cProbLog on top of ProbLog2 we adopt its syntax and use the `constraint/1` predicate to declare a constraint.

Example 4.6. *The last constraint of Example 4.5 can be declared in a cProbLog program*[2] *as:*

```
constraint(pack(boots) and exists X in {skis, board, helmet} :
       pack(X)).
```

△

[2] A cProbLog program is a program in the language cProbLog, that is a ProbLog program with constraints.

cProbLog supports constraints with more than one quantified variables. It also supports nested logic formulae. That is, formulae that contain subformulae with Boolean variables and quantifiers such that not all quantifiers appear in the left-hand side of the formula. In Example 4.7 we show such constraints.

Example 4.7. *Consider the following sentences in English and their translation in FOL and in cProbLog syntax.*

English:	*"Bill has at most one sister"*
FOL sentence:	$\forall x, y(SisterOf(x, Bill) \wedge SisterOf(y, Bill) \implies x = y)$
cProbLog constraint:	`for_all X of people(X), for_all Y of people(Y) : (sister_of(X, bill) and sister_of(Y, bill)) implies X == Y`
English:	*"Bill has exactly one sister"*
FOL sentence:	$\exists x(SisterOf(x, Bill) \wedge \forall y(SisterOf(y, Bill) \implies x = y))$
cProbLog constraint:	`exists X of people(X) : (sister_of(X, bill) and for_all Y of people(Y) : (sister_of(Y, bill) implies X == Y))`

Note that as domains are required in cProbLog we use a predicate `people/1` *that enumerates all people available in our (hypothetical) ProbLog program.* △

In a cProbLog program it is possible to declare more than one constraint. The set of all constraints $C = \{c_1, .., c_n\}$ in a cProbLog program defines a conjunction $\bigwedge_{i=1}^{n} c_i$ that needs to hold. That is, all constraints need to be satisfied. We ought to note that, while the declaration of multiple constraints implies their conjunction[3], the declaration of multiple queries (i.e., using the predicate `query/1`) has different meaning. In particular, every query q_i declared by `query(`q_i`).` is seen apart from the rest and ProbLog will compute the probability $P(q_i | \bigwedge_{i=1}^{n} c_i)$ for every query q_i. For simplicity, we use the denotation $P(q_i|C)$ instead of $P(q_i | \bigwedge_{i=1}^{n} c_i)$.

4.2.2 Semantics

The semantics of cProbLog is based on the semantics of ProbLog – a cProbLog constraint restricts the possible worlds of a ProbLog program.

A ProbLog program L defines a set of possible worlds $\Omega = \{\omega_1, .., \omega_m\}$. Each possible world $\omega_j \in \Omega$ specifies a unique truth value assignment for all ground probabilistic atoms. A specific truth value assignment of all ground probabilistic atoms determines the truth values of derived atoms, i.e., atoms that unify with

[3]The is valid for declaring evidence.

the head of a rule. Also each possible world ω_j is associated with a probability $P(\omega_j)$ such that $\Sigma_j^m P(\omega_j) = 1.0$.

A constraint c_i on a ProbLog program L is a logic formula over the atoms of L and is satisfied if there exists at least one possible world $\omega_j \in \Omega$ that makes c_i true; in case there is no such possible world c_i is not satisfied. A set of constraints $C = \{c_1, .., c_n\}$ on a ProbLog program L defines the conjunction $\bigwedge_{i=1}^n c_i$ that needs to hold. Let $\Omega^{c_i} \subseteq \Omega$ denote the set of possible worlds in which the constraint $c_i \in C$ is satisfied. The conjunction $\bigwedge_{i=1}^n c_i$ then holds for the possible worlds in the intersection $\Omega^C = \cap_{i=1}^n \Omega^{c_i}$. Naturally, this set of possible worlds is a subset of the possible worlds of the initial ProbLog program L: $\Omega^C \subseteq \Omega$.

Adding a set of constraints C to a ProbLog program L yields a cProbLog program L^C. Satisfying the constraints of L^C boils down to determining the set of possible worlds of L which make the constraints true, that is to determine $\Omega^C \subseteq \Omega$. Then the sum of the probabilities of all possible worlds that satisfy the constraints may be smaller than 1.0. Therefore we need to normalize the probability of each possible world that satisfies the constraints so that their sum equal one. A cProbLog program L^C defines a distribution over possible worlds as given by Equation 4.1.

$$P_C(\omega_i) = \begin{cases} \frac{P(\omega_i)}{\sum_{\omega_k \in \Omega^C} P(\omega_k)}, & \text{if } w_i \in \Omega^C \\ 0, & \text{if } \omega_i \in \Omega \setminus \Omega^C \end{cases}, \tag{4.1}$$

where $P(\omega_i)$ is the probability of possible world ω_i as defined in Chapter 2. The denominator which expresses a summation over the probabilities of all possible worlds that satisfy the constraints, is a normalization factor.

Example 4.8. *Consider the cProbLog program composed by the ProbLog program and the ground constraint of Example 4.4. The result of applying Equation 4.1 for each possible world of that program is shown in the following table:*

Possible World	*pack: skis*	*boots*	*board*	*helmet*	*Query inlimit(10)*	*Constraint*	$P(\omega)$	$P_C(\omega)$ $\omega \models C$
ω_1	*T*	*T*	*T*	*T*	*F*	*T*	*0.0017*	*0.0019*
ω_2	*T*	*T*	*T*	*F*	*F*	*T*	*0.0034*	*0.0038*
ω_3	*T*	*T*	*F*	*T*	*F*	*T*	*0.0116*	*0.0131*
ω_4	*T*	*T*	*F*	*F*	*T*	*T*	*0.0235*	**0.0266**
ω_5	*T*	*F*	*T*	*T*	*F*	*F*	~~*0.0050*~~	*0.0000*
ω_6	*T*	*F*	*T*	*F*	*F*	*F*	~~*0.0101*~~	*0.0000*
ω_7	*T*	*F*	*F*	*T*	*T*	*F*	~~*0.0347*~~	*0.0000*
ω_8	*T*	*F*	*F*	*F*	*T*	*F*	~~*0.0704*~~	*0.0000*
ω_9	*F*	*T*	*T*	*T*	*F*	*T*	*0.0087*	*0.0098*
ω_{10}	*F*	*T*	*T*	*F*	*F*	*T*	*0.0176*	*0.0200*
ω_{11}	*F*	*T*	*F*	*T*	*T*	*T*	*0.0606*	**0.0689**
ω_{12}	*F*	*T*	*F*	*F*	*T*	*T*	*0.1231*	**0.1399**
ω_{13}	*F*	*F*	*T*	*T*	*F*	*T*	*0.0260*	*0.0295*
ω_{14}	*F*	*F*	*T*	*F*	*T*	*T*	*0.0528*	**0.0600**
ω_{15}	*F*	*F*	*F*	*T*	*T*	*T*	*0.1819*	**0.2067**
ω_{16}	*F*	*F*	*F*	*F*	*T*	*T*	*0.3693*	**0.4197**
						Sum:	*0.88*	*1.0*

△

For a ProbLog program (no evidence, no constraints) the MARG probability of a query q is the sum of the probabilities of all possible worlds in which q is true. In cProbLog, satisfying a set of constraints C limits the set of possible worlds in which q is true to the ones where the conjunction $q \wedge C$ is true: $\Omega^{q \wedge C} = \Omega^q \cap \Omega^C$. We define the probability of a query q given the constraints C are satisfied as:

$$P(q|C) = \sum_{\omega_i \in \Omega^{q \wedge C}} P_C(\omega_i) = \frac{\sum_{\omega_i \in \Omega^{q \wedge C}} P(\omega_i)}{\sum_{\omega_i \in \Omega^C} P(\omega_i)} \tag{4.2}$$

Example 4.9. *In the table of Example 4.8 the probabilities of the possible worlds that entail the conjunction* `inlimit(10)`$\wedge$`pack(skis) implies pack(boots)` *are marked in bold. Applying Equation 4.2 on the table in Example 4.8, that is, summing up these numbers results in:*

$$P(\texttt{inlimit(10)}|\texttt{pack(skis) implies pack(boots)}) = 0.9218.$$

△

ProbLog uses Bayes' rule to compute the conditional probability of a query given evidence: $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$ (see Chapter 2, Section 2.1.3). Equation 4.2 also can be expressed using Bayes' rule: $P(q|C) = \frac{P(q \wedge C)}{P(C)}$, since the nominator expresses the sum of the probabilities of possible worlds where both constraints and queries are true ($\sum_{\omega_i \in \Omega^{q \wedge C}} P(\omega_i) = P(q \wedge C)$) and the denominator is the sum of all possible worlds where the constraints are true ($\sum_{\omega_i \in \Omega^C} P(\omega_i) = P(C)$). That is why semantically we see constraints as a generalization of evidence.

Even though constraints generalize evidence we impose one practical restriction. Namely, while cProbLog constraints are considered part of the definition of the model, we use evidence to encode observations to an already defined model, i.e., a ProbLog program.

4.2.3 The Restrictive Nature of cProbLog Constraints

A cProbLog program L^C is a ProbLog program L extended with FOL constraints $C = \{c_1, .., c_n\}$. L defines a set of possible worlds Ω together with their probabilities. Satisfying then the constraints focuses on a subset of these possible worlds: $\Omega^C \subseteq \Omega$. Then a (COND) probability of query is computed with respect to this restricted set of possible worlds. That is why we say that cProbLog constraints are *restrictive*. This stands in contrast with the way constraints are used to handle probabilistic knowledge in other constraint logic formalisms such as CLP($\mathcal{BN}$) [64]; there constraints are *generative*.

4.3 Inference with cProbLog

In Example 4.2, we showed that the same observations can be expressed either as a FOL constraint or as an adequate ProbLog predicate (`add_ev/0`) that defines the "evidence" that has to be true. After adding the predicate `add_ev/0` to the initial ProbLog program and imposing the evidence `add_ev`= *true*, ProbLog computes the conditional probability of the query given that `add_ev`= *true*. Since `add_ev` and the constraint validate the same possible worlds, the computed probability equals the conditional probability of the query given that the constraint is satisfied. The transformation of constraints into ProbLog rules and imposing relevant evidence is the basis of our inference implementation. We call this approach the *constraint-evidence* approach. The inference task it focuses on is the COND task, i.e., computing the conditional probability that a query (or a set of queries) is true given that the constraints are satisfied.

4.3.1 ProbLog Inference Pipeline

In Chapter 2 we presented the general scheme of a ProbLog inference pipeline. A ProbLog inference pipeline is a sequence of transformation steps (or components) that reduces the computationally expensive inference task into a simple weighted model counting (WMC) problem. The first component – the grounding component, generates a propositional instance of an initial ProbLog program L,

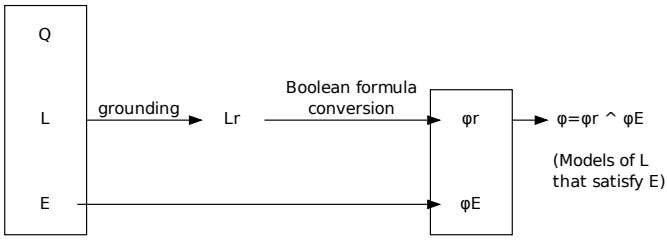


Figure 4.1: The steps to convert a ProbLog program L, together with a set of queries Q and evidence E into a Boolean formula.

relevant to a set of queries Q and evidence $E = e$. We denote this propositional instance L_r.

An atom is relevant with respect to Q and E if it occurs in some proof of a goal $g \in Q \cup E$. A ground rule is relevant with respect to Q and E if its head is a relevant atom and its body only contains relevant atoms. ProbLog uses SLD [38] or SLG [6][4] resolution on the logical part of L, that is, ignoring the label of probabilistic facts, resolution with the atoms in Q and E as its initial queries in order to collect all relevant ground atoms and ground rules. That is, it determines L_r. The next component is the Boolean formula conversion which converts L_r into a Boolean formula φ_r.

The conjunction φ_E of all the atoms in E with observed truth value *true* and the negation of atoms in E with observed truth value *false* states that the evidence should hold.

These transformations are illustrated in Figure 4.1.

ProbLog then uses the formula $\varphi = \varphi_r \wedge \varphi_E$ to generate a Boolean formula in negation normal form with special properties that allow to efficiently compute the conditional probabilities of the queries given the evidence [14].

cProbLog is based on the default ProbLog2 inference pipeline. That is why in the remaining of this chapter we focus on the implementation of ProbLog2, that is, we use a relevant ground logic program (LP) to represent the output of the grounding component and the formula computed by the Boolean formula conversion is output in Conjunctive Normal Form (CNF); we then compile this CNF into a formula in Smooth Deterministic Decomposable Negation Normal Form (sd-DNNF) [12].

[4]SLG resolution uses memoization of subgoals and allows to avoid querying the same atom twice.

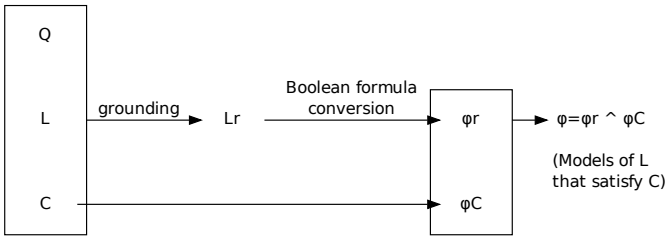


Figure 4.2: Convert a cProbLog program $L^C = L \cup C$, together with a set of queries Q into a Boolean formula.

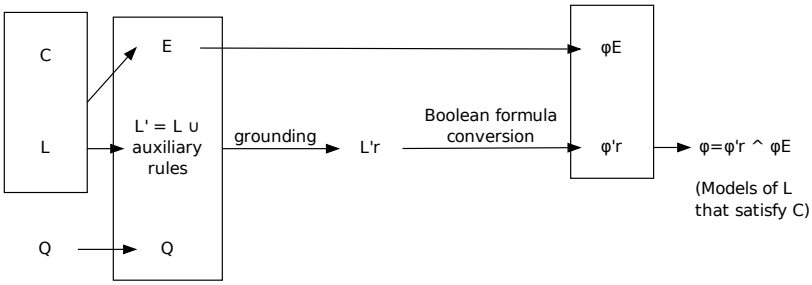


Figure 4.3: Convert a cProbLog program $L^C = L \cup C$, together with a set of queries Q into a Boolean formula that encodes the same possible worlds as L^C.

4.3.2 The Constraint-evidence Approach

Consider a cProbLog program L^C that extends a ProbLog program L with the set of constraints C ($L^C = L \cup C$). In order to compute the COND task for a set of queries Q given that the constraints C are satisfied it is required that ProbLog builds a Boolean formula that has the same possible worlds as L in which the constraints C are satisfied. That is, we need to apply a similar sequence of transformations as depicted in Figure 4.1 such that the models of φ correspond to the possible worlds of L in which the constraints C are satisfied. This is depicted in Figure 4.2

Instead of generating a ground LP relevant to the queries and the constraints, the idea of the constraint-evidence approach is to first convert the constraints into Prolog rules, then impose evidence on these rules and use the existing ProbLog inference pipeline to compute the COND task. This preprocessing is depicted in Figure 4.3.

The constraint-evidence approach augments the existing ProbLog inference pipeline with an additional preprocessing step that (i) converts the constraints to ProbLog rules; (ii) imposes the evidence that heads of these rules need to be true. Then by using the ProbLog inference pipeline to compute the COND task, in practice it computes the probability of the queries given that the constraints are satisfied.

The constraint-evidence approach adds a preprocessing step in the grounding component before the grounding commences, but leaves intact the rest of the pipeline[5]. Any ProbLog pipeline that can compute the COND task can be augmented to support constraints. It only needs the preprocessing step that converts constraint into Prolog rules and adds the evidence prior to grounding.

In the next section we prove correctness and illustrate the necessary transformations implemented by our method.

4.3.3 Correctness of the Constraint-evidence Approach

By imposing evidence on the Prolog rules that encode the constraints, L_r is in practice relevant to the queries and the constraints. The resulting set of possible worlds, implicitly encoded by the Boolean formula, is the correct set with respect to the semantics given in Section 4.2.2. First we investigate the simple case where the constraints are ground and in CNF; then we generalize.

Consider the simple case with no queries ($Q = \emptyset$) but only evidence (E), thus, L_r is relevant only to E. The models of φ satisfy the evidence. Consider a cProbLog constraint c that is ground and in CNF, we write it as φ_c. We can, by using the ProbLog approach, find the relevant grounding for each of the ground atoms of c and generate the corresponding CNF φ_r^c. The models of the conjunction $\varphi^c = \varphi_r^c \wedge \varphi_c$ are models of L that satisfy c. In the case that c and E contain exactly the same atoms and c enforces the same truth values as in E, $\varphi^c \Leftrightarrow \varphi$.

Instead of generating φ_r^c for the constraint c, we can also add the rule: `add_ev:-` c`.` to L and impose the evidence `add_ev`$= true$. `add_ev` is true iff the constraint formula c, i.e., `add_ev`$\Leftrightarrow \varphi_c$. ProbLog then will find the relevant instances of the rules for the ground atoms in c (as it does for evidence atoms). In the Boolean formula conversion step these instances will be converted into the Boolean formula $\varphi_r^{c'}$. Now we have $\varphi_r^{c'} \wedge$(`add_ev`$\Leftrightarrow \varphi_c$)$\wedge$(`add_ev`$= true$) which is equivalent to $\varphi_r^c \wedge \varphi_c$.

[5]In practice one can first invoke the constraint conversion in order to generate a ProbLog program from the initial cProbLog program and then use any ProbLog pipeline to compute the COND task.

For a set of variables $x_1..x_n$ with domains D_{x_1} to D_{x_n} and a FOL sentence

$$F_i(x_1, .., x_n):$$

1. Convert to Prenex Normal Form:
   $F_i(x_1, .., x_n) \overset{PNF}{\longrightarrow} F_i^{PNF}(x_1, .., x_n)$
2. Apply rewriting rules:

R1: $\exists x_i, D_{x_i} : F_i^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n) \rightarrow$
$F_i^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i/d_{x_i}, .., x_n) \vee$
$\exists x_i, D_{x_i} \setminus \{d_{x_i}\} : F_i^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n)$

R2: $\forall x_i, D_{x_i} : F_i^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n) \rightarrow$
$F_i^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i/d_{x_i}, .., x_n) \wedge$
$\forall x_i, D_{x_i} \setminus \{d_{x_i}\} : F_i^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n)$

R3: $F_i^{PNF}(x_1/d_{x_1}, .., x_n/d_{x_n}) \rightarrow$
`add_ev(i) :-` $F_i^{PNF}(x_1/d_{x_1}, .., x_n/d_{x_n})(\wedge/\texttt{,}, \vee/\texttt{;}, \neg/\texttt{\textbackslash+})$.

Figure 4.4: Transformation steps for constraint instantiation.

Generally, $Q \neq \emptyset$, and L_r also contains the relevant groundings for the queries $q_i \in Q$ which will appear in φ_r as is needed for correct weighted model counting.

When the constraints are not in CNF and are not ground, that is FOL sentences with quantified variables, they cannot be used as the body of a ProbLog rule. That is why in order to use the *constraint-evidence* approach, for each constraint that is not in CNF and is not ground we apply a set of transformations. It removes quantifiers from the constraints by instantiating each variable in the constraint. Note that the variables we quantify are required to have a domain. The transformation steps we employ in order to convert a single non ground constraint to Prolog rules are given in Figure 4.4.

Our method first makes sure that the FOL constraint is in prenex normal form [24] (PNF). In PNF all quantifiers are moved to the front of the FOL sentence, i.e., in front of the formula (variables and conjunctives). This form facilitates the application of our rewriting rules. Other systems that handle FOL constraints (e.g. IDP [48]) push quantifiers inwards to delay their instantiation. In IDP instantiation of the constraints is interleaved with their satisfaction. This allows to detect failure as soon as possible. In cProbLog we use SLG resolution in the grounding component to prove each query and evidence atom, including the ones derived from the auxiliary Prolog rules added during constraint instantiation. That is why the complete instance set needs to be collected first.

Once the FOL sentence is in prenex normal form then applying the rewrite rules generates ground quantifier-free formula. The first rewrite rule (see Figure 4.4), $R1$, transforms a sentence with an existentially quantified variable to a disjunction of all possible instances of the sentence that correspond to the values of the variable's domain. The second rule, $R2$, similarly generates a conjunction of all possible instances of the given sentence corresponding to the distinct values of the variable's domain. The last rewrite rule, $R3$, substitutes $\wedge$ with `,`, the $\vee$ with `;` and $\neg$ with `\+` and writes out a Prolog rule.

Proving the body of this rule during grounding is equivalent to a satisfiability check on the original FOL sentence. In order to check consistency with the formula relevant to the queries we rely on the Boolean formula – if the query and the constraints are inconsistent then the probability of their conjunction will be equal to zero.

For a set of constraints $C = \{c_1, .., c_n\}$ we use our rewrite rules in Figure 4.4 to convert each single constraint to a Prolog rule. That is, for each $c_i \in C$ there is a rule `add_ev(i) :-` $Body_i$ generated.

Example 4.10 shows the application of our approach on a small probabilistic graph (similar to Example 4.1). Once the constraints are converted to ProbLog clauses, we add the clause `evidence(add_ev(0), true).`. ProbLog then can apply its inference mechanism to compute the probability of a query given the evidence that the constraints are satisfied. In the process it finds the relevant ground program with respect to the query (`reach(c1, c3)`) and the evidence (`add_ev(0)`).

Example 4.10. *Consider the following cProbLog program:*

```
i_city(c1).    i_city(c2).
e_city(c2).    e_city(c3).
0.7::road(c1,c2).    0.8::road(c2,c3).    0.9::road(c1,c3).
reach(A,B):- road(A,B).
reach(A,B):- road(A,A1), reach(A1,B).

constraint(for_all X of i_city(X), exists Y of e_city(Y) :
           reach(X,Y)).
```

We apply our rewrite rules as follows:

1. *Initial constraint:*
   `for_all X of i_city(X), exists Y of e_city(Y) : reach(X,Y)`

2. *Apply R2:*
   `exists Y of e_city(Y) : reach(c1,Y)`$\wedge$
   `for_all X of i_city(X)`$_{X \neq c1}$`, exists Y of e_city(Y) : reach(X,Y)`
3. *Apply R1:*
   (`reach(c1,c2)`$\vee$`exists Y of e_city(Y)`$_{Y \neq c2}$`: reach(c1,Y)`*)*$\wedge$
   `for_all X of i_city(X)`$_{X \neq c1}$`, exists Y of e_city(Y) : reach(X,Y)`
4. *Apply R1:*
   (`reach(c1,c2)`$\vee$`reach(c1,c3)`*)*$\wedge$
   `for_all X of i_city(X)`$_{X \neq c1}$`, exists Y of e_city(Y) : reach(X,Y)`
5. *Apply R2, R1, R1:*
   (`reach(c1,c2)`$\vee$`reach(c1,c3)`*)*$\wedge$*(*`reach(c2,c2)`$\vee$`reach(c2,c3)`*)*
6. *Apply R3:*
   ```
   add_ev(0) :- (reach(c1,c2); reach(c1,c3)),
                (reach(c2,c2); reach(c2,c3)).
   ```

The constraint-evidence approach generates a ProbLog program from the cProbLog one:

```
i_city(c1).    i_city(c2).
e_city(c2).    e_city(c3).
0.7::road(c1,c2).    0.8::road(c2,c3).    0.9::road(c1,c3).
reach(A,B):- road(A,B).
reach(A,B):- road(A,A1), reach(A1,B).
add_ev(0) :- (reach(c1,c2); reach(c1,c3)),
             (reach(c2,c2); reach(c2,c3)).
evidence(add_ev(0), true).
```

Any query q that we post on this program will make ProbLog compute the conditional probability of q given that the constraint is satisfied. $\triangle$

Our constraint-evidence approach relies solely on the rewrite rules of Figure 4.4 which exhaustively generate all possible combinations of the domain elements. Its complexity is then $O(n^m)$, where n is the size of the (largest) domain and m is the number of variables for a single constraint.

4.3.4 Alternative Approach

An alternative method is to use directly the CNF of the constraints without first rewriting to ProbLog rules. Constraints in cProbLog are FOL sentences. As

such they are easily rewritten in CNF. Let L_r^Q be the relevant ground program with respect only to Q. Also, let φ_C be the CNF of the set of constraints C and φ_r^Q is the CNF of L_r^Q as computed by the Boolean formula conversion component in the ProbLog inference pipeline. Then we can use the conjunction $\varphi_r^Q \wedge \varphi_C$ to build the Boolean formula and compute the probabilities $P(q|C)$, for every $q \in Q$. We prefer the constraint-evidence approach for two reasons. First, we rely on the grounder of ProbLog to determine whether the constraints are satisfied and in case they are, which are the relevant ground atoms and rules that need to be added to the L_r^Q in order to determine the complete grounding of the input program. For example, in the Prolog rule generated from the constraint in Example 4.10 the atom `reach(c2, c2)` is not relevant and the grounding will determine only

```
add_ev(0) :- (reach(c1,c2); reach(c1,c3)), reach(c2,c3)).
```

as a relevant rule. Second, our approach retains the basic ProbLog pipeline and requires only a single preprocessing step to determine the instances of the constraints. In practice the implementation of our approach as a stand-alone code allows any ProbLog pipeline that can compute the COND task to use constraints.

4.3.5 Boundaries of the Approach

The constraint-evidence approach converts a non-ground FOL sentence into a ground formula in CNF – first the constraints are processed according to the rewrite rules in Figure 4.4 and a ProbLog program with evidence is generated from a cProbLog one; then the grounder of ProbLog determines the relevant part and the Boolean formula conversion computes a Boolean formula in CNF. The main drawback of our approach is the exhaustive instantiation of variables. To get a rough idea of the boundaries of the approach we run the constraint-evidence approach on a set of constraints with different number of variables and domain sizes and collect the run time of the transformation. The set up is as follows: (i) 4 different constraints c_i where $i = 2$ to 5 denotes the number of variables included in the constraint; (ii) each even variable is existentially quantified while each odd variable is universally quantified; (iii) each variable is associated with a domain of N elements (for simplicity in evaluation, the domains of different variables have the same sizes), where N ranges from 10 to 860. For example:
c_2 = `constraint(for_all X in {1 .. 100}, exists Y in {1 .. 100} : edge(X, Y) or edge(Y, X)).`
is a constraints with 2 variables and domains of size 100 and

c_3 = `constraint(for_all X in {1 .. 100}, exists Y in {1 .. 100}, for_all Z in {1 .. 100} : edge (X, Y) or edge(Y, X) or edge(X, Z).` is a constraint with 3 variables and domains of size 100. (iv) a maximum running time of 120 seconds is given for the constraint-evidence approach to process each constraint; (v) we run our experiments on a 4-core/8-thread Intel®Core™i7 @ 3400 MHz machine with 16GB RAM.

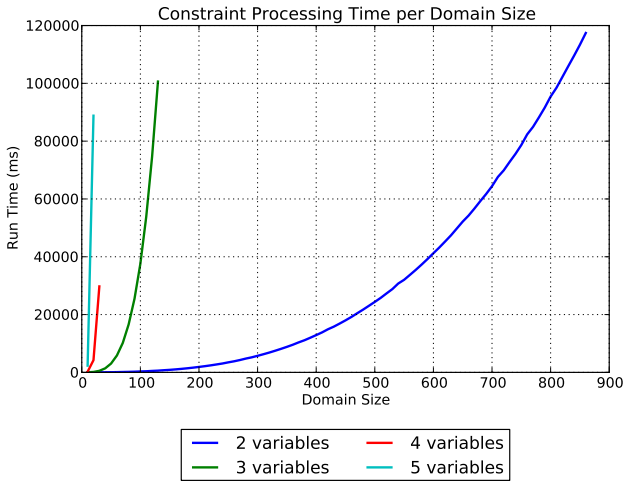


Figure 4.5: Run time for the constraint-evidence approach on constraints with growing number of variables and domain sizes.

Figure 4.5 summarizes the run times of the constraint-evidence approach per number of variables and domain size. It shows that our approach can process constraints with 2 variables and 860 elements in a domain and constraints over 5 variables and 20 domain elements. Still the exhaustive instantiation of the variables in a constraint can slow down the inference pipeline. In the next section we investigate approaches to optimize the constraint processing by reducing the number of instances.

4.4 Optimizing the Relevant Grounding for Constraints

In this section we present an optimization that applies on constraints and removes atoms that are relevant to the constraints but not to the queries. This optimization is inspired by the pattern-based compaction method presented in

Chapter 3. Theorem 4.1 and Theorem 4.2 state the conditions under which our optimization is correct with respect to the semantics of cProbLog; we use the notion of dependency set of a goal. The dependency set D_g of a goal g is the set of all ground atoms relevant to g, that is, these are all ground atoms (probabilistic or non-probabilistic) that appear in some proof of the goal g[6].

For a constraint c_i we define the dependency set D_{c_i} as the set of all relevant atoms of the goal `add_ev(i)`, where `add_ev(i)` is the head of the rules generated by the rewrite rules in Figure 4.4 for the constraint c_i.

Theorem 4.1. *Given a constraint c with dependency set D_c and a query q with dependency set D_q, such that $D_c \cap D_q = \emptyset$ then $P(q|c) = P(q)$.*

***Proof**: The proof is trivial and it follows from the statistical independence of events.*

$$
\begin{aligned}
p(q|C) &= \frac{p(q \wedge C)}{p(C)} \\
&= \frac{p(q) \cdot p(C)}{p(C)} \quad \textit{(because of statistical independence)} \\
&= p(q)
\end{aligned}
$$

■

Theorem 4.2. *Given a constraint $C = \bigwedge_{i=1}^{n} c_i$, with c_i, $1 \leq i \leq n$ a literal, and a query q such that C can be divided in two: $C = C^q \wedge C^*$, where (i) $C^q = \bigwedge_{i=1}^{m} c_i$, $1 \leq m \leq n$ has a dependency set D_{C^q} that is a subset of the dependency set of the query D_q, i.e., $D_{C^q} \subseteq D_q$; and (ii) $C^* = \bigwedge_{i=m+1}^{n} c_i$ has a dependency set D_{C^*} that does not intersect with the dependency set of the query, i.e., $D_{C^*} \cap D_q = \emptyset$. Then the probability of q given that the constraint C is satisfied $P(q|C)$ equals the probability of q given that only C^q is satisfied $P(q|C^q)$, i.e., $P(q|C) = P(q|C^q)$.*

***Proof**: Given that $C = C^q \wedge C^*$ then*

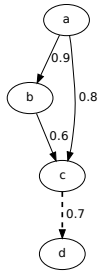
$$P(q|C) = P(q|C^q \wedge C^*)$$

The truth value of the conjunction C^* is independent from q or C^q. I.e., its conjuncts are neither in the dependency set of q (D_q) nor in the dependency set of C^q (D_{C^q}). Then because of the statistical independence $P(q|C^q \wedge C^*) = P(q|C^q)$. Therefore:

[6] In Chapter 2, Section 2.2.2 we give a detailed definition of relevant ground atoms and rules.

$$P(q|C) = P(q|C^q) \qquad \blacksquare$$

Example 4.11. *A ProbLog program encoding a probabilistic graph and a set of constraints. The solid edges are relevant to the query* `path(a, c).`*; the dashed one is not.*




```
0.9::edge(a, b).
0.8::edge(a, c).
0.6::edge(b, c).
0.2::edge(c, d).

path(A, B):-edge(A, B).
path(A, B):-edge(A, C),
            path(C, B).
query(path(a, c)).
```

Constraints:

c_1	$edge(c, d)$
c_2	$edge(a, b) \wedge edge(c, d)$
c_3	$edge(a, b) \vee edge(c, d)$

To compute the query q given that the constraints C are satisfied ($P(q|C)$), ProbLog uses Equation 4.2. The nominator and the denominator are sums over the probabilities of the possible worlds as defined in Equation 4.1.

Case 1: *Consider that only constraint c_1 is included in the cProbLog program. We use Theorem 4.1 and compute $P(q|c_1) = P(q)$.*

Case 2: *Consider now only c_2 is part of the cProbLog program. Then using Theorem 4.2 we can compute $P(q|edge(a, b) \wedge edge(c, d)) = P(q|edge(a, b)$.*

Case 3: *Consider only c_3 participates in the cProbLog program. The fact $edge(a, b)$ is relevant for the constraint and also for the query. The fact $edge(c, d)$ is not relevant to the query. We cannot remove any atoms from the ground LP similar to Case 2, as it this constraint forms a disjunction and therefore does not comply with the conditions in Theorem 4.2. This is shown by the following equation:*
First, we compute separately $P(q \wedge c_3)$ and $P(c_3)$ and then we divide; $p(e_{xy})$

denotes the probability of $edge(x, y)$*:*

$$
\begin{aligned}
P(q \wedge c_3) &= p(e_{ab})p(e_{ac})p(e_{bc})p(e_{cd}) + p(e_{ab})p(e_{ac})p(e_{bc})(1-p(e_{cd}))+ \\
&\quad p(e_{ab})p(e_{ac})(1-p(e_{bc}))p(e_{cd}) + p(e_{ab})p(e_{ac})(1-p(e_{bc}))(1-p(e_{cd}))+ \\
&\quad p(e_{ab})(1-p(e_{ac}))p(e_{bc})p(e_{cd}) + p(e_{ab})(1-p(e_{ac}))p(e_{bc})(1-p(e_{cd}))+ \\
&\quad (1-p(e_{ab}))p(e_{ac})p(e_{bc})p(e_{cd}) + (1-p(e_{ab}))p(e_{ac})(1-p(e_{bc}))p(e_{cd}) \\
&= (1-p(e_{ab}))p(e_{cd})\Big(p(e_{ac})p(e_{bc}) + p(e_{ac})(1-p(e_{bc}))\Big)+ \\
&\quad p(e_{ab})(1-p(e_{cd}))\Big((1-p(e_{ac}))p(e_{bc}) + p(e_{ac})(1-p(e_{bc})) + p(e_{ac})p(e_{bc})\Big)+ \\
&\quad p(e_{ab})p(e_{cd})\Big((1-p(e_{ac}))p(e_{bc}) + p(e_{ac})(1-p(e_{bc})) + p(e_{ac})p(e_{bc})\Big) \\
&= (1-p(e_{ab}))p(e_{cd})p(e_{ac})+ \\
&\quad p(e_{ab})(1-p(e_{cd}))(1-p(e_{ac}))p(e_{bc}) + p(e_{ab})(1-p(e_{cd}))p(e_{ac})+ \\
&\quad p(e_{ab})p(e_{cd})(1-p(e_{ac}))p(e_{bc}) + p(e_{ab})p(e_{cd})p(e_{ac}) \\
&= \Big((1-p(e_{ab}))p(e_{cd}) + p(e_{ab})(1-p(e_{cd})) + p(e_{ab})p(e_{cd})\Big)p(e_{ac})+ \\
&\quad p(e_{ab})(1-p(e_{cd}))(1-p(e_{ac}))p(e_{bc})+ \\
&\quad p(e_{ab})p(e_{cd})(1-p(e_{ac}))p(e_{bc})
\end{aligned}
$$

$P(c_3) = (1-p(e_{ab}))p(e_{cd}) + p(e_{ab})(1-p(e_{cd})) + p(e_{ab})p(e_{cd})$ *(Complete formula is omitted.)*

Dividing $P(q \wedge c_3)$ *to* $P(c_3)$ *and simplifying gives the following formula:*

$$\frac{P(q \wedge c_3)}{P(c_3)} = p(e_{ac}) + \frac{p(e_{ab})(1-p(e_{ac}))p(e_{bc})}{p(e_{ab}) + (1-p(e_{ab})p(e_{cd})}$$

The term $p(e_{cd})$ *remains in the denominator, as the probability* $P(q \wedge c_3)$ *is dependent on the probability of* $edge(c, d)$*.* △

Examples 4.11 illustrates the cases in which our constraint simplification is possible. Although its application may seem limited the combination with other optimizations (in particular the pattern based compaction) has a practical impact on the performance of cProbLog.

4.4.1 Implementation

We implemented our optimization method that detects and removes independent atoms from conjunctive constraints as a processing step that takes place after grounding is completed. It compacts the AND-OR graph associated to the relevant ground LP and the ground instances of the constraints. Our approach is similar to the Pattern-based compaction presented in Chapter 3.

In order to implement our method, first we modified the AND-OR graph generation method. Namely, we added an additional marker for every node in the AND-OR graph to keep track of the origin of that node. If a node is constructed from the ground program, we mark it with "grounding". If the node is build from the constraints then we mark it with "constraint". If an OR or a terminal node[7] needs to be added that already exists and is marked differently than the current origin, then its marker is substituted with "disjunction"; if the node is an AND node, then its marker is substituted with "conjunction". This extension of the AND-OR graph doesn't increase the memory complexity as it is just a different labeling of the nodes.

The algorithm we implement applies two steps: (i) detecting if the constraints and the queries are independent (Theorem 4.1), and if yes removal of the nodes and edges from the AND-OR graph originating from the constraints presented in Algorithm 3; and (ii) detection of conjuncts irrelevant from the query (Theorem 4.2) that are then removed form the AND-OR graph given in Algorithm 4. We implement an AND-OR graph as a collection of edges between nodes. We call an edge that connects a parent OR node with a child node an *OR edge* and an edge that connects a parent AND node with a child node an *AND edge*. Each edge is stored as a Prolog fact. Also terminal nodes are stored as facts.

Algorithm 3: Algorithm for detecting and removing independent constraints.

```
Data: An AND-OR graph
Result: An AND-OR graph
detect_independent_constraint ←
    \+ terminal_node(Terminal, _)/disjunction,
    \+ or_edge(OR, _)/disjunction,
    \+ and_node(AND, _)/conjunction.
simplify_graph_independent_constraint ←
    detect_independent_constraint,
    each(terminal_node(Terminal, _)/constraint,
       retract(terminal_node(Terminal, _)/constraint)),
    each(or_edge(OR, _)/constraint, retract(or_edge(OR, _)/constraint)),
    each(and_node(AND, _)/constraint, retract(and_edge(AND, _)/constraint)).
```

[7] Terminal nodes are a special kind of OR nodes (see Chapter 3).

Algorithm 4: Algorithms for detecting and removing of independent atoms from conjunctive constraints.

```
Data: An AND-OR graph
Result: An AND-OR graph
detect_independent_conjunction(NodeA, RefinedChildren) ←
    and_edge(NodeA, _)/constraint,
    all(Terminal, (
        and_edge(NodeA, Terminal)/constraint,
        terminal_node(Terminal, _)/constraint,
        ∄ or_edge(_, Terminal)/_
    ), Children),
    Children ≠ ∅,
    ∄ disjunctive_predecessors(NodeA),
    get_all_and_edge_sets(ChildSets),
    refine_cluster(ChildSets, Children, RefinedChildren).
disjunctive_predecessors(NodeB) ←
    or_edge(NodeA, NodeB).
disjunctive_predecessors(NodeB) ←
    and_edge(NodeA, NodeB),
    disjunctive_predecessors(NodeA).
update_graph(NodeA, Children) ←
    each(Child ∈ Children,
        (retract(terminal(Child, _)/constraint),
        retract(and_edge(NodeA, Child)/constraint))
    ),
    update_graph_empty_conjunction(NodeA).
update_graph_empty_conjunction(NodeA) ←
    all(Child, and_edge(NodeA, Child)/_, []),
    retractall(or_edge(ParentNode, NodeA)/_)).
```

Because we implement the AND-OR graph as a collection of edges our analysis of the complexity determines a bound with respect to the number of edges. We performed similar analysis for the detection and compaction algorithm in Chapter 3. For an arbitrary AND-OR graph G let us denote with $N_{or,constraint}, N_{or,grounding}, N_{or,disjunction}$ the number of *OR edges* that originate from the constraints, the ground LP or from both; with $N_{and,constraint}, N_{and,grounding}, N_{and,conjunction}$ the number of *AND edges* that originate from the constraints, the ground LP or from both; and with $N_{term,constraint}, N_{term,grounding}, N_{term,disjunction}$ the number of terminal nodes that originate from the constraints, the ground LP or from both, respectively. With N_{or}, N_{and} and N_{term} we denote all *OR edges*, *AND edges* and terminal nodes regardless the origin.

To detect whether a constraint[8] is independent from the query we first need to check all terminal nodes that originate from both the constraints and the ground LP; if there are no such nodes we need to look for OR edges that originate from both constraints and ground LP; and if there are no such edges

[8]Although we speak of one constraint, given the semantics of cProbLog a set of constraints forms a conjunction of all that constraints. That is why we can assume that there is only one constraint, without loss of generality.

we need to test the AND edges. That is done in a constant time: $O(1)$. To remove the subgraph associated with the constraint the complexity is: $O(N_{or,constraint} + N_{and,constraint} + N_{term,constraint})$.

In the detection step of Algorithm 4 we want to find out that we have a conjunction of literals that are terminal nodes and are not children of OR nodes. To do so we need to first collect all AND edges between an AND node that originates from the constraint and a terminal node, such that the terminal node does not participate in OR edges. Then we need to traverse the graph and check if any of the predecessors is an OR node. The total complexity to detect is $O(N_{and,constraint} \cdot N_{terminal,constraint} \cdot N_{or})$. To update the graph first we remove the edges from the detected node and its children. Then we also need to update the graph by removing all AND edges that point to non existing terminal nodes. We also need to remove any other edges that point to nodes that have already been removed. The complexity is: $O(N_{or,constraints} + N_{and,constraints} + N_{term,constraints})$.

4.4.2 Compatibility with other optimization approaches.

In Chapter 3 we presented a compaction method that detects 6 subgraph patterns in AND-OR graphs. We then use these patterns to compact the graph. This approach considers AND-OR graphs without additional markers. In order to use the compaction method of Chapter 3 together with the constraint optimization presented in this section a small change was required. We modified the detection and compaction algorithms to process AND-OR graphs with origin markers. That allowed us to use the pattern-based compaction approach alongside Algorithm 3 and Algorithm 4.

Grounding a ProbLog program together with a set of query and evidence atoms yields a ground logic program that is relevant to the queries and the evidence. Often the truth values specified by the evidence can be exploited in order to reduce the size of the relevant ground LP. In [14] we introduce a simple but efficient optimization of the grounding in which inactive rules are disregarded. A rule is considered inactive when a literal in its body is $false$ according to the given evidence in the ProbLog program. Consider a rule `r:-` B, where the body B is the conjunction $\texttt{b}_1$`, ..,` $\texttt{b}_n$. If at least one literal $\texttt{b}_i$ is false then B is also false. This statement is valid even if B contains probabilistic facts as well as in cases where any literal in B forms a cycle. If a literal $\texttt{b}_i$ causes a cycle then any proof of `r` which considers the body B is false (cf. [43]). That is, it is safe to remove inactive rules even if a literal in the body B causes a cycle.

The same intuition may apply also for disjunctive clauses given that the evidence determines a literal to be *true*. Given the disjunction $B =$ $\texttt{b}_1$`; ..;` $\texttt{b}_n$ it is

enough that one literal $\mathtt{b}_i$ is true for the whole disjunction B to be true. Then B can be substituted by the logical value *true*. A rule that has the disjunction B as a body, i.e., `r:-` B`.` can be replaced by `r:-` *true*. Such optimization is only valid when the literal $\mathtt{b}_i$ can be proven *true*. Otherwise it creates inconsistency and the inference fails.

If though a literal in body of conjunctions (disjunctions) is defined by the evidence to be true (false) it cannot be removed from the body as it may form a positive cycle and therefore needs to be kept.

This optimization considers only rules with conjunctive bodies, i.e., r`:-` $a_1, .., a_n$.. A rule r is inactive if at least one literal a_i in its body is false. An atom a_i can be defined false (or its negation defined true) by the evidence. Also, if there is no rule to prove r then the derived atom r is also false. Then any rule that contains the atom r in its body is also inactive. This way the information about false atoms is propagated during grounding. This optimization can reduce up to 17% of the relevant ground LP which may lead to up to 90% reduction of the Boolean formula [14]. Propagation of false evidence can be used together with the removal of independent constraint atoms, but some conditions need to be observed. Let us assume a ProbLog program that contains constraints but also is given evidence, that is, a cProbLog program with evidence. Also let evidence and constraints be consistent with the ProbLog program, but contradictory. Then the program is inconsistent and the probability of the queries is undefined or assumed 0.0. Propagating false evidence before optimizing the constraints is required. The constraint optimization removes independent atoms from conjunctions. It may be the case that these atoms cause the evidence and the constraints to be contradictory and once removed the program is provable. That is why first the complete ground LP needs to be generated – grounding queries, constraints and evidence, and then any optimization should take place.

Since in Section 4.2.2 we proved the equivalence between constraints and evidence then we can employ the optimization of independent atoms also for evidence – e.g., if the evidence states that a head of a rule is true then all its body is true and the optimization of independent atoms can be applied on the conjunction of the atoms in the body. Moreover, we can also use the inactive rule propagation for atoms declared to be false by the constraints.

4.4.3 Experiments

We tested the effects of our approach on 3 benchmark sets of cProbLog programs. Each of these sets contains two different types of programs corresponding to the best and the worst case scenarios for our algorithm. The benchmarks encode probabilistic graphs similar to the “Grid” benchmark (see Chapter 2).

```
0.5::edge(n_1_1, n_2_1).
0.5::edge(n_2_1, n_3_1).
0.5::edge(n_2_2, n_3_2).
0.5::edge(n_3_2, n_3_3).
0.5::edge(n_3_1, n_3_2).
0.5::edge(n_2_1, n_2_2).
0.5::edge(n_1_1, n_2_2).
0.5::edge(n_2_2, n_3_3).
```

a.

```
0.5::edge(n_1_1, n_2_1).
0.5::edge(n_2_1, n_3_1).
0.5::edge(n_2_2, n_3_2).
0.5::edge(n_3_2, n_3_3).
0.5::edge(n_3_1, n_3_2).
0.5::edge(n_2_1, n_2_2).
0.5::edge(n_1_1, n_2_2).
0.5::edge(n_2_1, n_3_2).
0.5::edge(n_2_2, n_3_3).
```

b.

```
0.5::edge(n_1_1, n_2_1).
0.5::edge(n_2_1, n_3_1).
0.5::edge(n_1_2, n_2_2).
0.5::edge(n_2_2, n_3_2).
0.5::edge(n_1_3, n_2_3).
0.5::edge(n_2_3, n_3_3).
0.5::edge(n_1_1, n_1_2).
0.5::edge(n_1_2, n_1_3).
0.5::edge(n_2_1, n_2_2).
0.5::edge(n_2_2, n_2_3).
0.5::edge(n_3_1, n_3_2).
0.5::edge(n_3_2, n_3_3).
0.5::edge(n_1_1, n_2_2).
0.5::edge(n_1_2, n_2_3).
0.5::edge(n_2_1, n_3_2).
0.5::edge(n_2_2, n_3_3).
```

c.

Figure 4.6: Probabilistic graphs encoded as ProbLog programs to use for experimenting with the optimization for independent atoms; $m = 3$.

Grid 1 A rooted directed acyclic graph (DAG) with nodes $\{n_{x,y} : x = 1, .., m, y = 1, .., m, y \leq x\}$, with $m = 3, .., 25$. Edges connect node $n_{x,y}$ to $n_{x+1,y}$ and to $n_{x,y+1}$; also node $n_{x,x}$ to $n_{x+1,x+1}$. Figure 4.6.a shows such a graph encoded as a ProbLog program. There are 23 programs in a benchmark set – one for each value of m.

Grid 2 A rooted directed acyclic graph (DAG) with nodes $\{n_{x,y} : x = 1, .., m, y = 1, .., m, y \leq x\}$, with $m = 3, .., 25$. Edges connect node $n_{x,y}$ to $n_{x+1,y}$, to $n_{x,y+1}$ and to $n_{x+1,y+1}$. Figure 4.6.b shows such a graph encoded as a ProbLog program. There are 23 programs in a benchmark set – one for each value of m.

Grid 3 A rooted directed acyclic graph (DAG) with nodes $\{n_{x,y} : x = 1, .., m, y = 1, .., m\}$, with $m = 3, .., 25$. Edges connect node $n_{x,y}$ to $n_{x+1,y}$, to $n_{x,y+1}$ and to $n_{x+1,y+1}$. Figure 4.6.c shows such a graph encoded as a ProbLog program. There are 23 programs in a benchmark set – one for each value of m.

For each benchmark we use the following set of rules, also referred to as background knowledge (BK):

```
path(A, B):- edge(A, B).
path(A, B):- edge(A, A1), path(A1, B).
```

and a query which looks for the path between node $n_{1,1}$ and $n_{k,k}$, where $k = \lfloor m//2 \rfloor$.

The data in our benchmarks form grids, as illustrated in Figure 4.7.

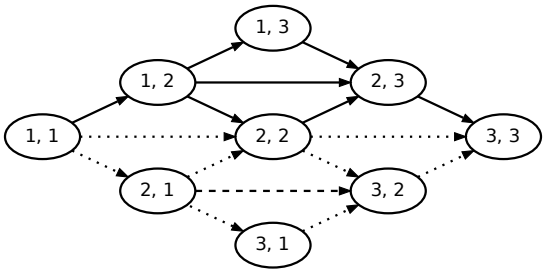


Figure 4.7: A grid representing the data encoded by the benchmark programs used in the experiments. The fine dotted edges represent a "Grid 1" benchmark of 3×3 nodes (see Figure 4.6 a)); the fine dotted edges together with the dashed edge represent a "Grid 2" benchmark of 3×3 nodes (see Figure 4.6 b)); and the whole grid represents a "Grid 3" benchmark of 3×3 nodes (see Figure 4.6 c)).

For each benchmark program we generate two separate cProbLog programs with the following:

Existential:
```
constraint(exists XY of domain_XY : XY = X/Y and edge(X, Y)).
```
Universal:
```
constraint(for_all XY of domain_XY : XY = X/Y and edge(X, Y)).
```

We use one variable, namely `XY` that unifies with a pair `X/Y` and the predicate `domain_XY` to enumerate the values of all such pairs. These are all pairs of nodes in the given program between which an edge exists. When grounded the first constraint will generate a disjunction of all edges; the second will generate a conjunction which can be simplified according to our method. The first constraint is the worst case for our approach. It requires to search through the subgraph associated with the constraint and will not detect anything. The second constraint is the best case scenario, where we remove all nodes and edge from the graph that do not originate from the ground LP.

We ran cProbLog using the c2d compiler and a timeout of 540.0 seconds on each benchmark program. We ran each program 5 times and like for other experiments, we consider the average of the following measurements: (i) total run time; (ii) number of CNF variables; (iii) number of CNF clauses; (iv) number of sd-DNNF nodes and (v) number of sd-DNNF edges. We managed to run with at least one of the settings (optimized or non-optimized): all 25 benchmark programs with both existentially and universally quantified constraints for Grid 1; 23 of the programs with existentially quantified constraints and the

25 programs with universally quantified constraints for Grid 2; and 19 of the programs with existentially quantified constraints and the 25 programs with universally quantified constraints for Grid 3. Our results are summarized in Figure 4.8 and Figure 4.9. Each line in a diagram presents the ratio between a measurement when the optimization is enabled and when it is disabled. E.g., let's denote with T_{tot} the total time when no optimization is used and with T_{tot^+} the total time when the optimization is used; the value we draw is $t = \frac{T_{tot^+}}{T_{tot}}$; when $t < 1$ then cProbLog with our optimization outperform cProbLog with no optimization. The value of these ratios is given on the $y-$axis in logarithmic scale; on the $x-$axis we enumerate the cProbLog programs in an incremental order according to the AND-OR graph size.

The diagrams show that: (i) for programs containing constraints with universally quantified variables that are grounded to a conjunction of atoms, our optimization reduces the total run, the CNF variables and clauses and the sd-DNNF nodes and edges; (ii) for existentially quantified constraints that are grounded to disjunctions of atoms and no optimization can be performed, our algorithms do not increase the run time noticeably. Hence, we can state that constraint optimization by removing independent atoms from conjunctions should be performed by default during cProbLog inference.

The benchmarks we used are the two extreme cases – when nothing can be compacted or when the whole subgraph that originates from the constraints (but not from the query) can be compacted. In practice a "standard" benchmark will not comply with this conditions. Since the time for detection in the cases when nothing is compacted is insignificant (see Figure 4.9) we are sure that applying our constraint compaction method for all types of cProbLog programs could only be beneficial.

4.5 Examples

The cProbLog programs in this section aim at (i) familiarizing the user with writing constraints in cProbLog; and (ii) showing the equivalence with evidence in ProbLog, where feasible we give the equivalent evidence atoms as it would have to be written in ProbLog.

4.5.1 Probabilistic Graph 1

In Figure 4.10 we show a program encoding a very small probabilistic graph with a query for path existence between nodes `1` and `3`. This example shows

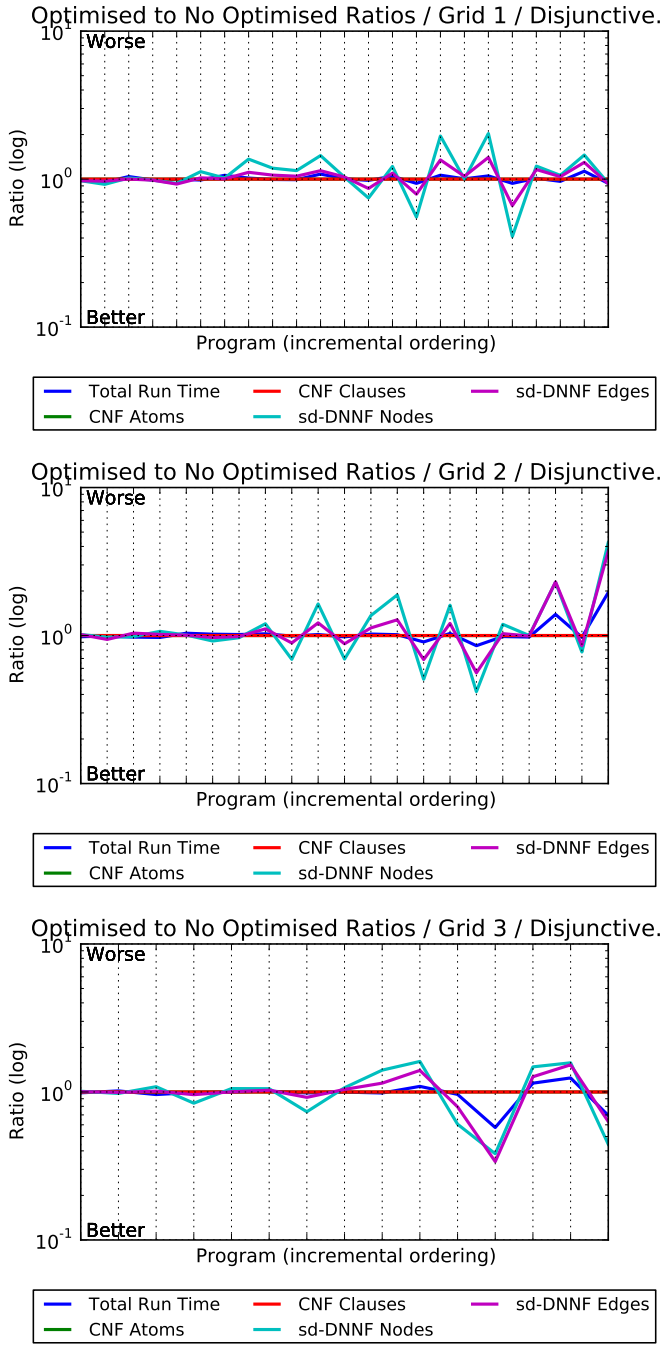


Figure 4.8: Experimental results comparing the total run time, the number of CNF variables and clauses and the number of sd-DNNF nodes and edges when enabling or not the optimization of Algorithm 4. Existentially quantified variables in constraints.

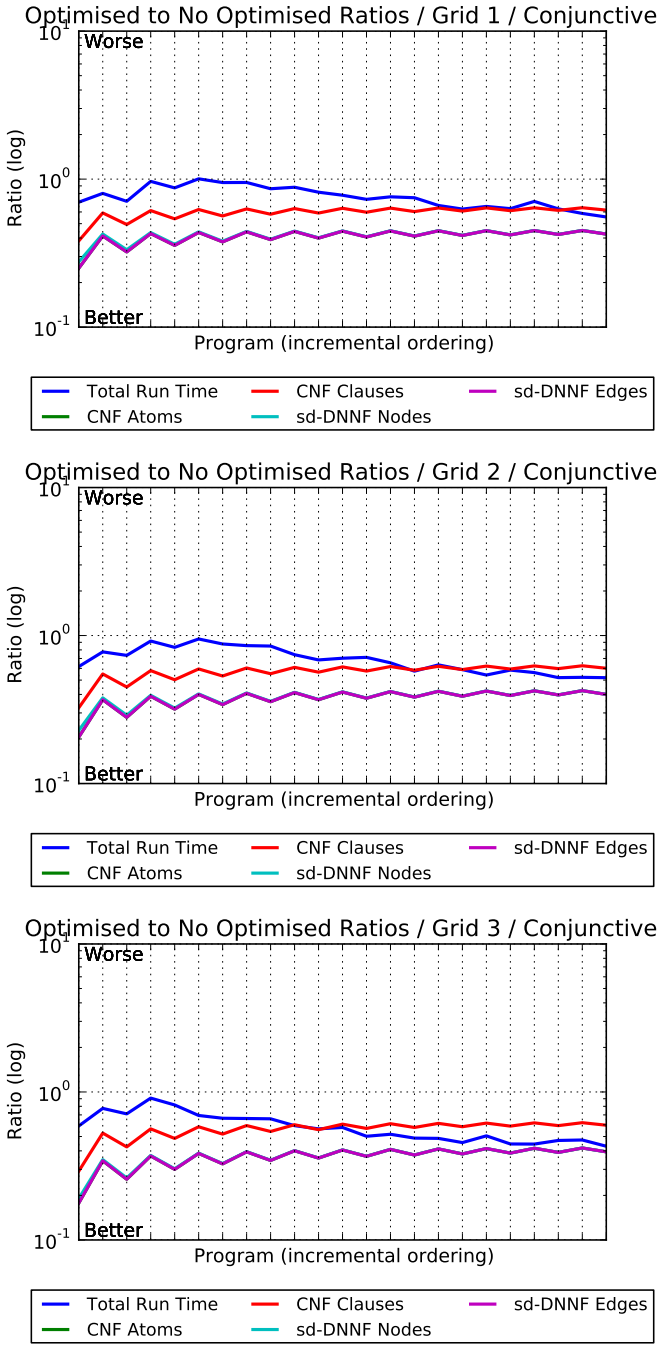


Figure 4.9: Experimental results comparing the total run time, the number of CNF variables and clauses and the number of sd-DNNF nodes and edges when enabling or not the optimization of Algorithm 4. Universally quantified variables in constraints.

```
0.7::edge(1, 2).    0.8::edge(1, 3).    0.4::edge(2, 3).
path(A, B):- edge(A, B).
path(A, B):- edge(A, A1), path(A1, B).

query(path(1, 3)).
constraint((exists X in {1, 2}, exists Y in {2, 3}) :
            X \== Y and not edge(X, Y)).
```

Figure 4.10: A small probabilistic graph encoded as a cProbLog program.

a constraint over probabilistic facts. The domains for the variables in the constraint are given as sets of values. As they overlap, the constraint defines that X is different from Y. The MARG probability of the query (without the constraint being considered) is $P(path(1,3)) = 0.856$. The constraint c encodes that at least one edge is false. This makes invalid one of the possible worlds in which the query is true, namely, the one where all three edges are true. This world has the probability 0.224. To compute the probability $P(path(1,3)|c)$ we take the sum of the probabilities of worlds where $path(1,3) \wedge c$ is true $(0.632 = 0.856 - 0.224)$ and renormalize (apply Bayes' rule) over the ones where the constraint c is true $(0.776 = 1.0 - 0.224)$, see Equation 4.2 in Section 4.2.2. The result is $P(path(1,3)|c) = 0.814$.

For this constraint there is no straightforward alternative using evidence.

4.5.2 Probabilistic Graph 2

The program presented in Figure 4.11 encodes a probabilistic graph where each edge has a length of 1. The `path/3` predicate finds a path between two nodes, as well as its length. We use constraints to add knowledge about the path length(s). E.g., one can query for the probability of a path from `a` to `h` when only considering paths longer than 5. ProbLog computes the conditional probability of the query given that the constraints are satisfied: $P(path(a,h)|C) = 0.014$. In this example we use a set to define the domain of `X`. For this example, instead of the constraint one can use the following evidence:

```
evidence(path(a, h, 1), false).
evidence(path(a, h, 2), false).
evidence(path(a, h, 3), false).
evidence(path(a, h, 4), false).
evidence(path(a, h, 5), false).
```

```
0.6::edge(a,b).       0.7::edge(a,c).       0.55::edge(b,c).
0.36::edge(b,d).      0.45::edge(c,e).      0.7::edge(d,f).
0.8::edge(e,d).       0.25::edge(e,g).      0.25::edge(f,g).
0.3::edge(f,h).       0.4::edge(g,h).
path(A, B):- path(A, B, _).
path(A, B, 1):- edge(A, B).
path(A, B, L):- edge(A, B1), path(B1, B, L1), L is L1 + 1.

constraint(for_all L in {1,2,3,4,5} : not path(a, h, L)).

query(path(a, h)).
```

Figure 4.11: A ProbLog program encoding a probabilistic graph with path length restrictions.

4.5.3 Burglary-earthquake-alarm Bayesian network

Figure 4.12 illustrates a cProbLog program which encodes a Bayesian network [14]. The program defines two neighbors – John and Mary. Burglary and earthquake may trigger an alarm. If the alarm goes off, one of the people calls the owner of the house. The initial program states that either John or Mary or both may call. Using a constraint we encode that at most one of them calls. ProbLog can handle multiple queries simultaneously. In ProbLog2 we use the predicate `query/1` to state a query (see Section 4.2.1). We state multiple queries by multiple `query/1` declarations as shown in this example. ProbLog then will compute the probability that each query[9] is true given that the constraints are satisfied: $P(\texttt{burglary}|C)$ and $P(\texttt{earthquake}|C)$.

For this simple example we can easily write a predicate which is equivalent to our rule rewriting transformation but much more simple as shown in Figure 4.12 b) and c). The observed blow-up is due to the fact that variables are blindly instantiated with values from their domain, without evaluating any subgoals. The evaluation is (currently) left to ProbLog.

Despite this drawback, it can easily be noticed that with growth of the domain such manual encoding becomes infeasible. Moreover, the rules which cProbLog generates can be preprocessed and optimized.

[9]Recall from Section 4.2.1 that while the declaration of multiple constraints imply their conjunction, multiple `query/1` declarations state separate queries that are not related.

```
%ORIGINAL PROGRAM:
person(john).
person(mary).
0.1::burglary.
0.2::earthquake.
0.7::hears_alarm(X) :- person(X).

alarm:- burglary. alarm:- earthquake.
calls(X):- alarm, hears_alarm(X).

%CONSTRAINTS:
constraint((for_all X of person(X),
  for_all Y of person(Y)) :
  (calls(X) and calls(Y))
  implies X == Y).

%QUERIES:
query(burglary).
query(earthquake).
```

a. A cProbLog program.

```
add_ev(0):-
  ((\+ calls(mary); \+ calls(mary));
    mary==mary),
  (\+ calls(mary); \+ calls(john));
    mary==john)),
  ((\+ calls(john); \+ calls(mary));
    john==mary),
  (\+ calls(john); \+ calls(john));
    john==john)).
evidence(add_ev(0), true).
```

b. cProbLog transformation of the constraint.

```
ev1:-
   \+ calls(mary).
ev1:-
   \+ calls(john).
evidence(ev1, true).
```

c. User-defined evidence.

Figure 4.12: An example program of a Bayesian network.

4.5.4 Student exams

The example in Figure 4.13 is about students and exams. There are four ways that a student passes an exam: by having studied enough; by luck; by cheating and by knowledge from previous experience. Passing an exam has certain probabilities, eg., a student passes an exam by luck with probability 0.4. One can query this program for the probability a student passes all exams. The first constraint is a ground constraint which expresses that one can pass "Machine Learning" or "Artificial Neural Networks (ANNs)" by luck but not both of them. The second constraint defines that 3 years of experience are not enough to pass any exam. For these constraints there is no straightforward alternative using evidence.

4.6 Comparison to Other Probabilistic Constraint Logics

We compared cProbLog to other Constraint Probabilistic Logic Programming formalisms – PCLP [51], CLP($\mathcal{BN}$) [64] and CHRiSM [77] with respect to defining and using constraints.

```
0.8::pass_exam(Exam, studied_enough).
0.4::pass_exam(Exam, luck).
0.7::pass_exam(Exam, cheating).

P::pass_exam(Exam, experience, Years):- P is Years/7.
exam(E):- member(E, [prolog, cog_sci, anns, comp_vis, m_learn]).
student(john, [prolog, cog_sci, anns, comp_vis, m_learn], 3).
succeed(Student):- student(Student, Exams, Experience), pass_all(Exams, Experience).
pass_all([], _).
pass_all([F|Rest], Exp):- pass_one(F, Exp), pass_all(Rest, Exp).
pass_one(Ex, _):-pass_exam(Ex, _).
pass_one(Ex, Years):-pass_exam(Ex, experience, Years).

constraint(pass_exam(m_learn, luck) implies not pass_exam(anns, luck)).
constraint((for_all X of exam(X), for_all Y in {0,1,2,3}) : not pass_exam(X, experience, Y)).

query(succeed(john)).
```

Figure 4.13: The "student exams" example program.

4.6.1 cProbLog and PCLP

The language PCLP [51] (short for Probabilistic Constraint Logic Programming) combines constraint logic programming with probabilistic inference. A PCLP theory $\mathcal{T}^{PCLP}$ is defined by a set of constraints $\mathcal{C}_\mathcal{T}$, a set of random variables $\mathcal{V}_\mathcal{T}$ and a set of rules $\mathcal{R}_\mathcal{T}$. $V(t_1,..,t_n) \sim \{p_1 : c_1,..,p_m : c_m\}$ defines the random variable $V(t_1,..,t_n) \in \mathcal{V}_\mathcal{T}$ (with t_i a term), over the distribution[10] $\{p_1 : c_1,..,p_m : c_m\}$, where c_j is a constraint and p_j its probability. The constraints c_1 to c_m specify the possible values of the variable and p_1 to p_m – the probability of their assignment.

Example 4.12. *PCLP random variables are multi-valued and can be assigned sets or intervals of values specified by the constraints:*

- $X \sim \{0.1 : 0 \leq X \leq 1, 0.3 : 1 \leq X \leq 2, 0.6 : 2 \leq X \leq 3\}$ – *the random variable* X *has a value in the interval* $[0, 1]$ *with probability* 0.1*, with probability* 0.3 *in the interval* $[1, 2]$ *and with* 0.6 *in the interval* $[2, 3]$
- $Patrol(0) \sim \{0.5 : m, 0.5 : l\}$ *illustrates a random variable* $(Patrol(0))$ *with two possible values* m *and* l *and their corresponding probabilities* 0.5 *and* 0.5 *(see Example 4.17).*
- $Yield(apple) \sim N(12000.0, 1000.0)$ *exemplifies the use of normal distribution with mean* 12000 *and variance* 1000*.*

△

[10]The distribution can also be continuous eg., $\mathcal{N}(\mu, \sigma^2)$

A PCLP rule is valid only when the constraints in its body are satisfied. In practice, a rule propagates the distributions which satisfy the constraints. A PCLP theory $\mathcal{T}$ is provable in a class of distributions. For a ground atom q these distributions specify an interval $[P_{min}(q), P_{max}(q)]$, such that $P_{min}(q) \leq P(q) \leq P_{max}(q)$ for each member P of the class of distributions.

Example 4.13 [51] shows a PCLP theory which we can query for the probability range of q, i.e., $[P_{min}(\mathtt{q}), P_{max}(\mathtt{q})]$.

Example 4.13. *For the PCLP theory:*

$X \sim \{0.1 : 0 \leq X \leq 1, 0.3 : 1 \leq X \leq 2, 0.6 : 2 \leq X \leq 3\}$
$Y \sim \{0.1 : 0 \leq Y \leq 1, 0.3 : 1 \leq Y \leq 2, 0.6 : 2 \leq Y \leq 3\}$
$\mathtt{q} \leftarrow Y < 0.75$
$\mathtt{q} \leftarrow Y < 1.25, 0.25 * X + Y < 1.375$

there are 9 *possible worlds, given that there are* 3 *choices for the variable* X *and* 3 *choices for the variable* Y*. In* 5 *of them the query* q *could be true; in the other* 4 q *is false. The probability range of* q *is between* $[0.01, 0.22]$*.* △

From a certain perspective a PCLP theory is very similar to one defined under CLP(FD) [7] or CLP(R) [27]: constraints specify sets of discrete constants or intervals of real numbers and are used to determine the possible values of a variable. Rules define the dependency among constraints[11].

With respect to the inference, the key difference between cProbLog and PCLP is that while cProbLog aims at computing the marginal probability of a query given that the constraints are satisfied, PCLP aims to compute the lower and upper bounds $P_{min}(q)$ and $P_{max}(q)$.

PCLP is a more expressive language than ProbLog. Under certain conditions though, a PCLP theory can be mapped to a ProbLog program with annotated disjunctions (Chapter 5). However, there is no *direct* correspondence to a cProbLog program. In contrary, PCLP can be augmented with cProbLog constraints.

4.6.2 cProbLog and CLP($\mathcal{BN}$)

CLP($\mathcal{BN}$) [64] is a probabilistic extension of constraint logic programming. In logic programming existentially quantified variables are presented by terms build from Skolem functors. In analogy to Probabilistic Relational Models (PRM) [17]

[11] In contrast, the constraints in a cProbLog program rule out some of the possible worlds of that program.

CLP($\mathcal{BN}$) uses a Bayesian network to represent the joint probability distribution over such (Skolem) terms.

A CLP($\mathcal{BN}$) program $L^{CLP(BN)}$ is a set of clauses ($\mathtt{C}$= $\{\mathtt{C}_1, .., \mathtt{C}_n\}$) of the form $H \leftarrow A/B$. H is a literal called the head, A is (a possibly empty) conjunction of literals and B is a (possibly empty) conjunction of constraints. H and A define the logic part of the clause and B – its probabilistic part. Each constraint has the form $\{V = Rv$ with $CPT\}$ and identifies the possible values of V with the random variable Rv. The conditional probability table CPT defines a probability distribution over the possible values of Rv. Each random variable is a Skolem term and appears in only one clause[12] of $L^{CLP(BN)}$ and is linked to a set of variables, e.g., $sk_i(X_1, .., X_n)$ is a Skolem term, where sk_i is its functor and $X_1, .., X_n$ are distinct variables which appear outside the term. The constraints in CLP($\mathcal{BN}$), similarly to PCLP and in contrast to cProbLog, define the possible values of a variable (V) through the probability distribution of the random variable.

Each clause $\mathtt{C}_i \in \mathtt{C}$, associated to a set of probability distributions defines a Bayesian network BN_i. The nodes of BN_i are labeled by variables or Skolem terms. For a given subset of clauses, the constraints create dependencies between the Bayesian networks associated with these clauses to generate a large Bayesian network BN. A CLP($\mathcal{BN}$) program $L^{CLP(BN)}$ defines a unique joint probability distribution over the ground Skolem it contains as the Bayesian network BN. It generates a constraint network for a query which when evaluated gives the marginal probability distribution of that query.

CLP($\mathcal{BN}$) targets Probabilistic Relational Models. It is especially suitable to program Bayesian networks. In Example 4.14 we show a part of a CLP($\mathcal{BN}$) program encoding a Bayesian network which expresses the relations between students, grades, courses and professors [64].

Example 4.14. *The following CLP($\mathcal{BN}$) clause states that a student can be considered intelligent with probability* 0.7 *(and not intelligent with* 0.3*):*

```
student_intelligence(S, Int) :-
    {Int = i(S) with p([h, l], [0.7, 0.3], [])}.
```

According to the previous clause all students have the same probability for being intelligent. Each student, though, should be considered apart. This can be expressed as:

```
student_intelligence(S, Int) :-
```

[12]Follows from the definition of Skolemization.

```
    int_table(S, Values, IDist),
    {Int = i(S) with p(Values, IDist, [])}.

int_table(alice, [h, l], [0.4, 0.6]):-!.
int_table(bob, [h, l], [0.7, 0.3]):-!.
...
```

The next clause encodes the conditional probabilities of the grades given the course difficulty and the intelligence of the student.

```
grade(Reg, Grade):-
    registration(Reg, Course, Student),
    course_difficulty(Course, Dif),
    student_intelligence(Student, Int),
    {Grade = grade(Reg) with p([a, b, c, d],
                               [0.8, 0.6, 0.0, 0.4, 0.3, 0.0,
                                0.1, 0.3, 0.1, 0.3, 0.3, 0.0,
                                0.1, 0.0, 0.5, 0.2, 0.3, 0.5,
                                0.0, 0.1, 0.4, 0.1, 0.1, 0.5],
                               [Dif, Int])}.
```

△

CLP($\mathcal{BN}$) supports also meta predicates. The clause in Example 4.15 combines `setof/3` to *aggregate* all grades of a student with the build-in `average/3` to generate a CPT (conditional probability table) for the ranking of the student.

Example 4.15. *To aggregate all grades of a student in CLP($\mathcal{BN}$) we can use the metapredicatae* `setof/3`.

```
student_ranking(S, Rank) :-
     setof(Grade, C^(registration(C,S), grade(C, Grade)), Grades),
     average([], Grades, CPT),
     {Rank = ranking(S) with CPT}.
```

△

Example 4.15 shows a metacall on the constraints in `registration/2` and `grade/2`. That is, a new constraint `{Rank = ranking(S) with CPT}.` is defined by means of other constraints (in `registration/2` and `grade/2`). In contrast the default ProbLog implementations (ProbLog1 and ProbLog2) do not support metacalls. MetaProbLog [45] is a ProbLog implementation that support metacalls and metapredicates.

Example 4.16. *In ProbLog one can use a set of probabilistic facts* `0.4::student_intelligence(alice).`, `0.7::student_intelligence(bob).` *to encode the same knowledge as encoded by the* `student_intelligence/2` *predicate of Example 4.14. For more flexibility one can use* intentional *probabilistic facts:*

```
P::student_intelligence(S):-
    intelligence(S, P).

intelligence(alice, 0.4).
intelligence(bob, 0.7).
...
```

△

With the core ProbLog language one cannot directly encode multivalued random variables (as the one expressed by the constraint in `grade/2` in Example 4.14) but requires annotated disjunctions which we discuss in Section **??**. Annotated disjunctions are multiheaded disjunctive clauses which specify mutually exclusive choices [87, 50, 86]. Example 4.17 compares CLP($\mathcal{BN}$) and ProbLog with annotated disjunctions on encoding a Hidden Markov Model [64].

Example 4.17. *Consider a Hidden Markov Model that illustrates the shifts of two police officers – Manissian (who is careful) and Lufy (who is a bit lax). An officer who is patrolling at day T also will patrol at day $T+1$ with probability* 0.8*; with probability* 0.2 *at day $T+1$ it will be another policeman to patrol.*

CLP$\mathcal{BN}$ [64]:

```
patrol(0, P):-!,
    {P = p(0) with p([m, l], [0.5, 0.5], [])}.
patrol(T, P):-
    T1 is T - 1, patrol(T1, P0),
    {P = p(T) with p([m, l], [0.8, 0.2,
                              0.2, 0.8], [P0])}.
```

The CPT used in the first clause of the predicate `patrol/2` *assigns the probability* 0.5 *for the event that Manissian (m) is a patrolling officer on the first day (day* 0*) as well as* 0.5 *for the event that Lufy (l) patrols on day* 0*. In the second clause the CPT gives the probability of one of the officers patrolling on the current day (*`T`*) conditioned on the previous day (*`T-1`*), i.e., who was patrolling the previous day. In particular, the first and the second values are the probabilities that today*

Manissian or Lufy will patrol given that Manissian was patrolling yesterday; the third and the fourth values are the probabilities that Manissian or Lufy will patrol today given that yesterday it was Lufy to patrol.

In ProbLog we use annotated disjunctions to encode the conditional events and the CPTs:

```
0.5::patrol(m, 0); 0.5::patrol(l, 0) <- true.
0.8::patrol(m, T); 0.2::patrol(l, T) <- T>0, T1 is T-1,
                                        patrol(m, T1).
0.2::patrol(m, T); 0.8::patrol(l, T) <- T>0, T1 is T-1,
                                        patrol(l, T1).

evidence(patrol(m, 4), true).
query(patrol(m, 10)).
```

△

A CLP($\mathcal{BN}$) program defines a set of probability distributions over the models of the underlying logic program. Computing the probability of a query given evidence in the framework of CLP($\mathcal{BN}$), is solving the Bayes' network generated from the conjunction of the network corresponding to the query and the one corresponding to the evidence.

Example 4.18. *(Continuing Example 4.17) A spy wants to bypass the guard at day* 10*. He has observed that at day* 4 *Manissian was patrolling. So, he needs to know what is the probability that Lufy, who is easier to bypass, will patrol at day* $T^* = 10$ *given his observation. Given the query*

`:- patrol(Officer, 10), patrol(m, 4).`*, CLP($\mathcal{BN}$) computes:*

```
p(Officer=m)=0.523328,
p(Officer=l)=0.476672
```

△

A cProbLog program defines a probability distribution over possible worlds. Determining the truth values of a set of atoms when satisfying a constraint, restricts the set of possible worlds. A query is computed over the valid (with respect to the constraints) possible worlds after normalizing their probability according to Equation 4.2.

We compared CLP($\mathcal{BN}$) and cProbLog from the perspective of constraints. There are three features of CLP($\mathcal{BN}$) language which mainly distinguish it from cProbLog: (i) meta calls; (ii) continuous probability distributions and (iii) the generative nature of CLP($\mathcal{BN}$) constraints in comparison to the restrictive cProbLog constraints. Because CLP($\mathcal{BN}$) supports (i) and (ii) while ProbLog does not[13], we can classify CLP($\mathcal{BN}$) a more expressive language than ProbLog.

4.6.3 cProbLog and CHRiSM

CHRiSM [77] (short for CHance Rules induce Statistical Models) is a probabilistic logic programming language which extends PRISM [66] with Constraint Handling Rules [18, 19]. Constraint Handling Rules is a high level language based on multiheaded rewrite rules and PRISM is a probabilistic logic programming language and a system. PRISM is similar to ProbLog with annotated disjunctions (as defined in Section **??**). It employs *switches* [66] to encode random events. The main difference between ProbLog and PRISM is that repeated calls to the same switch (in PRISM) is interpreted as an independent random event in contrast to repeated calls to an annotated disjunction (in ProbLog)[14].

A CHRiSM program consists of a set of rewrite rules of the form `P ?? Hk \ Hr <=> G | B`, called chance rules. In a chance rule, `P` is a probabilistic expression, `Hk` is a conjunction of kept head constraints, `Hr` a conjunction of removed head constraints, `G` is a guard condition (a Prolog goal to be satisfied) and `B` is the body of the rule. The probabilistic expression `P` can be a number (similar to the label of probabilistic facts in ProbLog), an arithmetic expression (like the probability of intentional probabilistic facts in ProbLog), an experiment name or even omitted. The rule body `B` is defined as a conjunction of CHRiSM constraints, Prolog goals and/or probabilistic disjunctions. A probabilistic disjunction is either an annotated disjunction or a CHRiSM-style disjunction of the form `P ?? D1 ; .. ; Dn` where `P` is an experiment name determining the probability distribution.

A CHRiSM constraint `c(X1, .., Xn)` is a Prolog-like predicate where `c` is its predicate and its arguments `X1` to `Xn` are Prolog terms. cProbLog constraints are FOL formulae. The initial point of a CHRiSM program (the query) is a set of constraints $\mathcal{S}$ called a store. Applying exhaustively the rules of the program results in a new store called the answer or the result. A rule can be applied if there is a matching substitution which unifies a subset of constraints from $\mathcal{S}$

[13]The extensions Hybrid ProbLog [22] supports continuous distributions.

[14]More details about the differences and similarities between PRISM and ProbLog can be found in [30].

to the head of that rule and also, the guard `G` is satisfied. Depending on the probability expression of the rule it can be either applied (chosen) or ignored (disregarded). An example of CHRiSM rules is given in Example 4.19 [77].

Example 4.19. *The following CHRiSM rule generates a random graph where each edge is considered with probability* 50%*:*

```
0.5 ?? node(A), node(B) ==> edge(A,B).
```

It uses a number to define the probability of applying the rule. The next rule is an example of using expression to define the probability of the rule:

```
eval(3/(N-1)) ?? nb_nodes(N), node(A), node(B) ==> edge(A,B).
```

The probability of the rule is defined according to the number of selected nodes. △

When a rule is applied all the constraints matching `Hr` are removed from $\mathcal{S}$ (the ones in `Hk` are kept), the goals in `B` are called and the constraints in `B` are added to the store. This generates a new store $\mathcal{S}'$ thus modifying the program state. The *execution state* is the state of the program after applying a specific rule. The goal is reached by a chain of rule applications, defining a sequence of states. The *transition* between states is labeled with a probability. The probability of the final state (or the goal) is then the product of each transition's probability. The rule chain is generated by a random walk in the directed graph defined by the transitions.

In (c)ProbLog the probability of a query considers all possible worlds defined by the random variables (the probabilistic atoms) of the program in which a query is true. From a certain perspective we can call CHRiSM constraints *restrictive* – when a set of constraints are satisfied it determines which (ground) rules may apply and the rest to be ignored. In the cProbLog interpretation of *restrictive* constraints when a constraint is satisfied it rules out a subset of the possible worlds which are generated from the initial ProbLog program.

4.6.4 Integrating cProbLog

Because cProbLog is built as a transformation step independent from the hosting system it is very easy to integrate cProbLog functionality with other Probabilistic Logic systems besides ProbLog. We have chosen to show the integration of cProbLog constraints into CLP($\mathcal{BN}$) although integrating cProbLog is also straightforward for many of the other probabilistic logic formalisms, such as PCLP, PRISM, MetaProbLog, etc.

To support cProbLog constraints we do not change CLP($\mathcal{BN}$) architecture. The only thing we need to do is to call the cProbLog instance generator (i.e., the constraint processing step of our constraint-evidence approach) in the CLP($\mathcal{BN}$) program. Invoking the cProbLog grounder will collect all cProbLog constraints and instantiate them, generating Prolog clauses. Giving the evidence that these clauses are true (see Section 4.3.3) will trigger SLD resolution to determine whether the constraints are satisfied. It will then compute the probability of the query given (the evidence that) the constraints are satisfied according to the CLP($\mathcal{BN}$) method. We illustrate this extension in Example 4.20.

Example 4.20. *The following program is an extension of the patrolling officer example (Example 4.15) with cProbLog constraints:*

```
:-ensure_loaded('cProbLog_grounder').
patrol(0, P):-!,
    {P = p(0) with p([m, l], [0.5, 0.5], [])}.
patrol(T, P):-
    T1 is T - 1, patrol(T1, P0),
    {P = p(T) with p([m, l], [0.8, 0.2,
                              0.2, 0.8], [P0])}.

constraint(exists X in {1, 3}: patrol(m, X)).

%CALL TO cProbLog
:- cproblog_for_clpbn.
```

At the end, after all declarations in the program we use the call to the cProbLog instantiation in order to process the constraints and construct a CLP($\mathcal{BN}$) program with evidence instead of constraints. △

4.7 First Order Logic and ProbLog

We ought to note that cProbLog is not the only system that combines First Order Logic (FOL) and ProbLog. [4] presents a language of probabilistic FOL formulae, called FOProbLog, defines its semantics and presents an approach for translating FOProbLog programs to ProbLog programs. This translation then allows to use ProbLog inference to compute a lower and upper bounds for the probability of a query formula.

An FOProbLog theory encodes a set of FOL sentences[15] of the form:

$$\phi = \forall \bar{x}.\psi_1 : \alpha_1 \vee .. \vee \psi_n : \alpha_n$$

where $\bar{x}$ is a set of variables, ψ_i with $i = 1..n$ is a FOL formula over some of the variables in $\bar{x}$, and α_i is the probability of the formula ψ_i ($\alpha_i > 0$). Each sentence ϕ expresses an independent belief about the world. The formulae $\psi_1, .., \psi_n$ are the possibilities for ϕ; each ψ_i can be believed with probability α_i and is exclusive from the rest ψ_j with $j = 1, .., n$ and $j \neq i$.

From a language perspective FOProbLog and cProbLog have some differences. On the one hand, cProbLog augments ProbLog2 with additional constructs to declare constraints. These constraints are FOL sentences over ground or non ground, probabilistic or non-probabilistic atoms; they do not include probabilistic labels. On the other hand the FOProbLog language considers FOL sentences as language constructs; some of them may also express probabilistic facts. Furthermore, a disjunction in FOProbLog expresses mutually exclusive possibilities while in cProbLog a disjunction over formulae is true if at least one (but also more) of these formulae is true. Both cProbLog and FOProbLog use domains to restrict the set of possible groundings.

An FOProbLog theory $\mathcal{T}^{FOProbLog}$ defines belief sets constructed by adding one sentence ϕ at a time. Belief sets correspond to total choices of a ProbLog program, as defined in Chapter 2. That is for a sentence ϕ with two possibilities and for each existing belief set there will emerge two more belief sets each of which will include the one or the other possibility of ϕ. Some belief sets may be inconsistent and therefore should have probability 0. FOProbLog defines probability distributions over consistent belief sets. Hence for each such set its probability needs to be normalized. Similarly in cProbLog we need to normalize with the sum of the probabilities of the possible worlds in which the constraints hold. This operation (for both FOProbLog and cProbLog) corresponds to conditioning.

While in cProbLog all constraints form a single conjunction that needs to hold, the FOL sentences in $\mathcal{T}^{FOProbLog}$ can be seen as individual constraint. Any distribution that satisfies them is a model. Then, each consistent belief set defined by $\mathcal{T}^{FOProbLog}$ may extend to more than one distribution and therefore defines an interval of the minimum and the maximum probability that a query q is true: $[P_{min}(q), P_{max}(q)]$. This semantics are more close to the semantics of PCLP rather than the semantics of cProbLog. FOProbLog uses ProbLog

[15]The syntax of the FOProbLog language is different from the presented form of FOL sentences. It contains probabilistic facts as in ProbLog (or cProbLog) and FOL formulae. Here we do not discuss the syntax further and refer the interested reader to [4] for details.

inference to determine this interval by employing Stickel's transformation [78] to convert from FOProbLog language to core ProbLog language. cProbLog also converts to core ProbLog language. cProbLog uses the rewrite rules in Figure 4.4.

4.8 Conclusion

The cProbLog language, initially introduced in [15], is a Constraint Probabilistic Logic Programming formalism which enriches ProbLog with constraints. Constraints are a generalization of evidence to FOL sentences. Each constraint is true in a subset of the possible worlds of the ProbLog part of the program. Satisfying all constraints of a cProbLog program requires to find the possible worlds where the conjunction of these constraints is true. Satisfying constraints restricts the set of possible worlds in which a query (or a set of queries) is true. When computing the probability of a query, given the constraints are satisfied we renormalize over the possible worlds restricted by the constraints.

In this chapter we described the first implementation of cProbLog. Our algorithm converts a constraint to a ground ProbLog rule and imposes evidence on it. With the existing inference mechanisms of ProbLog the conditional probability of a query given this evidence is the same as the conditional probability of the query given that the constraints are satisfied.

Our approach is built on top of ProbLog's inference mechanism without changing it. It can be easily employed by other systems. A drawback of our current implementation is the rather naive grounding of formulae with quantifiers (see Figure 4.12). Then we presented an approach to improve the grounding by considering only relevant constraint instances. In the future we aim to improve cProbLog by directly merging the CNF of the relevant ground program (with respect to a set of queries and evidence atoms) with the CNF of the constraints in order to bypass the Boolean formula conversion step for constraints.

We also compared cProbLog to other Constraint Probabilistic Logic Programming formalisms from the perspective of type and usage of constraints. Also we incorporated cProbLog with CLP($\mathcal{BN}$).

Chapter 5

ProbLog Programs with Annotated Disjunctions

Probabilistic facts provide a rather simple encoding of basic randomness – random events that can be either true or false, e.g. the availability of a direct road between two cities or a tossed coin shows heads. Various probabilistic inference and learning tasks with probabilistic facts are relatively easy to define and implement. From a modeling perspective though, it is often convenient to use annotate disjunctions [87, 50, 86]. An annotated disjunction encodes a set of atoms from which at most one is true at the same time. Annotated disjunctions are particularly convenient for expressing multi-valued random variables, e.g., rolling a die. Annotated disjunctions increase the expressivity of a probabilistic logic language, e.g., LPADs [87], CP-Logic [50].

ProbLog already supports annotated disjunctions by encoding them as probabilistic facts and rules which retain the mutual exclusiveness defined by the annotated disjunction by enforcing negation of some probabilistic facts. This encoding was implemented to handle annotated disjunctions for the MARG task and subsequently for the COND task [21]. For the MPE and MAP tasks, supported in ProbLog2, the current encoding is no longer correct. These tasks have application in problems like diagnosis and prognosis and their importance motivates our research to provide correct and efficient inference. In this section we present an encoding based on cProbLog constraints (Section 4.1) which maps annotated disjunctions to probabilistic facts and Prolog rules similar to the encoding of [21], but in contrast it uses constraints to ensure the mutual exclusiveness defined by the annotated disjunctions.

The work described in this chapter adds a new encoding for annotated disjunctions to ProbLog. This leads to the expansion of the expressive power of the ProbLog language with annotated disjunctions not only for the MARG and COND inference tasks, but also for MPE and consequently the MAP inference tasks. We also implemented the MPE inference task for ProbLog.

5.1 Annotated Disjunctions

5.1.1 Syntax

An annotated disjunction (AD) a is a multi-headed rule of the form

$$p_1 :: h_1; ..; p_n :: h_n \leftarrow b_1, .., b_m.$$

where p_i is the probability of the head atom h_i, and the conjunction of the literals $b_1, .., b_m$ forms the body of the AD. We denote with $head(a)$ the set of head atoms of the AD a and with $body(a)$ the set of body literals of a. The sum of the probabilities of all head atoms of an AD is smaller or equal to 1: $\sum_{h_i \in head(a)} p_i \leq 1$. If the body is true, the AD probabilistically *causes* one of the head atoms to become true, otherwise the AD does not have an effect. Thus, if the same atom appears in the heads of multiple ADs, they correspond to different causes, e.g., a window can break because a stone was thrown at, but also because of an earthquake. If $\sum_{i=1}^{n} p_i < 1$ there is a probability $(= 1 - \sum_{i=1}^{n} p_i)$ that none of the head atoms is caused to be true. As this can be made explicit by adding an extra *none* head atom, we assume that probabilities sum to one.

Example 5.1. *Consider a game with a bag containing red, green and blue balls. This knowledge is expressed by AD* a_1*. A player randomly decides to pick a ball (with probability* 0.6*) or not (with probability* 0.4*), encoded by AD* a_2*:*

a_1*:* `0.6::red(b1); 0.3::green(b1); 0.1::blue(b1) <- pick(b1).`
a_2*:* `0.6::pick(b1); 0.4::no_pick(b1) <- true.`

The knowledge encoded by AD a_2 *can be expressed also by a probabilistic fact, eg.,* `0.6::pick(b1).` *which implies* `not pick(b1).` *with probability* 0.4. △

For an AD a, the atoms in $head(a)$ can be ground or non ground. Similar to the intentional probabilistic facts (see Chapter 2) the non ground atoms in the head of a correspond to a set of ground atoms. That is, an annotated disjunction with a head that contains non ground atoms corresponds to a set of ground annotated disjunctions, as shown in Example 5.2. That is why our discussion in the remaining of this section focuses on ground ADs.

Example 5.2. *The annotated disjunction*

a: `0.6::red(B); 0.3::green(B); 0.1::blue(B) <- pick(B).`

expresses that for each possible instance $\mathtt{b}_i$ *of the variable* `B` *there is an annotated disjunction*

a_i*:* `0.6::red(`$\mathtt{b}_i$`); 0.3::green(`$\mathtt{b}_i$`); 0.1::blue(`$\mathtt{b}_i$`) <- pick(`$\mathtt{b}_i$`).` △

5.1.2 Semantics

We now summarize the process semantics of annotated disjunctions as developed in CP-logic. Note, that there are two different semantics defined for annotated disjunctions: [87] defines the semantics of ADs in the context of logic programs with annotated disjunctions (LPADs) and later [85, 86] define it from the perspective of *Causal-Probabilistic events* (CP-events in short) as the *process semantics*. Equivalence of the two semantics with respect to interpretations of the theory is proven in [86].

Definition 5.1 (Probability Tree)**.** *Let* $A = \{a_1, \ldots, a_k\}$ *be a set of ground annotated disjunctions over atoms* L_A*. A* probability tree $Tr(A)$ *is a tree where every node* n *is labeled with an interpretation* $I(n)$ *assigning truth values to a subset of* L_A *and a probability* $P(n)$*, constructed as follows:*

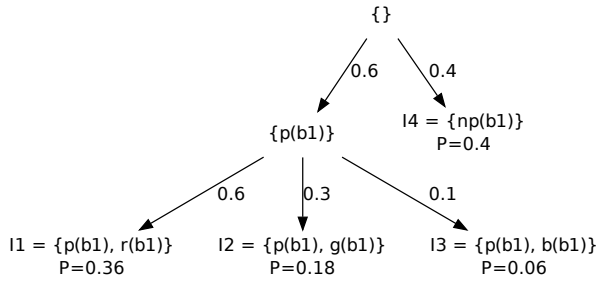
- *The root node* $\bot$ *has probability* $P(\bot) = 1.0$ *and interpretation* $I(\bot) = \{\}$*.*
- *Each inner node* n *is associated with an AD* a_i *such that*
  - *no ancestor of* n *is associated with* a_i*,*
  - *all positive literals in body*(a_i) *are true in* $I(n)$*,*
  - *for each negative literal* `\+`l *in body*(a_i)*, the positive literal* l *cannot be made true starting from* $I(n)$*.*

  and has one child node for each atom $h_j \in head(a_i)$*. The* j^{th} *child has interpretation* $I(n) \cup \{h_j\}$ *and probability* $P(n) \cdot p_j$*, where* p_j *is the probability of the head atom* h_j*.*
- *No leaf can be associated with an AD following the rule above.*

The path from the root to a leaf n *is called a* selection σ_n *with probability* $P(\sigma_n) = P(n)$*. We say that each selection* σ_n *defines an interpretation* $I(\sigma_n)$ $(I(\sigma_n) = I(n))$*. The probability of an interpretation* I *of* L_A *is the sum of the probabilities of all leaves* n *in the tree with* $I(n) = I$*.*

All probability trees for a given set of ADs A define the same distribution over selections [86]; from here on, we refer to an arbitrary probability tree when mentioning "the probability tree of A".

Example 5.3. *A probability tree associated with the ADs of Example 5.1 and its selections are:*



Selection:	*Interpretation*	*Probability* $P(\sigma_i)$
σ_1	$I1 = \{$`pick(b1), red(b1)`$\}$	0.36
σ_2	$I2 = \{$`pick(b1), green(b1)`$\}$	0.18
σ_3	$I3 = \{$`pick(b1), blue(b1)`$\}$	0.06
σ_4	$I4 = \{$`no_pick(b1)`$\}$	0.4

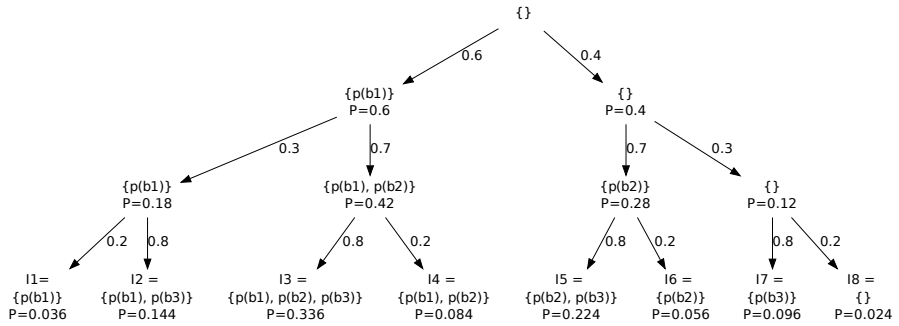
Here, all selections define different interpretations. △

We can also express the semantics of ProbLog as a probability tree. That is, we can build a probability tree from the set of probabilistic facts in a ProbLog program such that (i) its leaves correspond to the possible worlds of the initial ProbLog program and (ii) the probabilities of the leaves equal the probabilities of the possible worlds. A possible world is defined by the ground probabilistic facts of the ProbLog program (see Chapter 2). That is why we build the probability tree from the set of probabilistic facts and not from the set that also contains derived atoms. An interpretation, encoded in the leafs, will represent a partial model of the ProbLog program that does not include the derived atoms; it can be extended to a complete model by including derived atoms that are true according to the possible world. We illustrate this in Example 5.4.

Example 5.4. *For the set of probabilistic facts:*

```
0.6::pick(b1).
0.7::pick(b2).
0.8::pick(b3).
```

we can build a probability tree starting from an empty interpretation and adding one fact after another – once when the fact is considered true and once when the fact is considered false:



Each interpretation contains only the facts that are true; the rest of the probabilistic atoms that are not in the interpretation are false. △

5.2 ProbLog Encoding of Annotated Disjunctions

A ProbLog program with annotated disjunctions combines the expressive power of both ADs and probabilistic facts. To support MARG inference for ProbLog programs with annotated disjunctions [21, Chapter 3] introduces a method that translates ADs to a set of probabilistic facts with normalized probabilities (according to Equation 5.1) and a Prolog rule for each of its head atoms, where the bodies of these rules are mutually exclusive.

More precisely, for each annotated disjunction

$$a_j = p_1 :: h_1; ..; p_n :: h_n \leftarrow b_1, .., b_m.$$

this method first adds a set of probabilistic facts

$$\{p'_1 :: pf(j, 1, vars(a_j)), \ldots, p'_{n-1} :: pf(j, (n-1), vars(a_j))\}$$

where $vars(a_j)$ denotes all variables that appear in a_j; and second it extends the ProbLog program with the rules[1]:

$$h_1 \text{ :- } b_1, \ldots, b_m,\ pf(j, 1, vars(a_j)).$$

$$h_2 \text{ :- } b_1, \ldots, b_m,\ pf(j, 2, vars(a_j)),\ \backslash+\ pf(j, 1, vars(a_j)).$$

$$\ldots$$

$$h_n \text{ :- } b_1, \ldots, b_m, \backslash+\ pf(j, n-1, vars(a_j)), \ldots,\ \backslash+\ pf(j, 1, vars(a_j)).$$

And the probability p'_i is defined as:

$$p'_i = \begin{cases} p_i & \text{if } i = 1 \\ \frac{p_i}{1 - \sum_{j=1}^{i-1} p_j} & \text{if } i > 1 \end{cases} . \tag{5.1}$$

When an AD a_j is ground ($vars(a_j) = \emptyset$), the third term of the probabilistic fact pf is empty and can be simplified to $p'_i :: pf(j, i)$.

A detailed description of the method, which we refer to in the remaining of this chapter as the *ProbLog encoding* can be found in [21, Chapter 3] and a proof of correctness for the task of computing the marginal probability of a query is given in [21, Appendix A]. Example 5.5 illustrates its application.

Example 5.5. *The ProbLog encoding for Example 5.2 is:*

```
0.6::pf(1, 1).
0.75::pf(1, 2).
red(b1):- pick(b1), pf(1, 1).
green(b1):- pick(b1), pf(1, 2), \+ pf(1, 1).
blue(b1):- pick(b1), \+ pf(1, 2), \+ pf(1, 1).

0.6::pf(2, 1).
pick(b1):- pf(2, 1).
no_pick(b1):- \+ pf(2, 1).
```

△

Essentially, mutual exclusiveness is made explicit through the `pf/2` facts in the bodies.

[1] If the sum of the probabilities of the head atoms does not equal 1.0 there is a probability that nothing is selected - the *none* choice. If that is the case, another probabilistic fact should be added: $pf(j, n, vars(r_j))$ with the corresponding probability as calculated with Equation 5.1. Also, the fact should be added in the body of the rule generated from the encoding for the last head atom.

5.3 Most Probable Explanation for ProbLog Programs

The inference task that has received most attention by the PLP community is computing the marginal probability of a ground query atom q, the MARG task, and its generalization the COND task, that computes the probability of a query q given evidence $E = e$. In PLP literature the MARG task is also referred as the success probability of the query.

In statistical relational learning (SRL) [20] and probabilistic graphical models (PGM) [37] one of the key tasks is to find the most likely state of the world where a set of observations (the evidence) holds, also called MPE inference. ProbLog aims to bridge the gap between PLP and SRL by supporting tasks common for both fields.

Formally, the Most Probable Explanation (MPE) task, is defined as follows.

Definition 5.2 (Most Probable Explanation)**.** *Given a probability distribution $P(V)$ over a set of discrete random variables V and a truth value assignment e to a subset of random variables $E \subseteq V$ (the evidence), the task of finding the most probable explanation (MPE) is to determine a truth value assignment u to the remaining random variables $U = V \setminus E$ with maximal probability, that is,* $\text{MPE}(E) = \arg\max_u P(U = u \mid E = e)$.

A solution of the MPE task is also called an MPE state. Solution techniques developed for MPE include methods based on knowledge compilation [10], integer linear programming [82], and weighted MAX-SAT [61].

For a **ProbLog program without ADs**, V is the set of ground atoms – probabilistic and derived, $E = e$ fixes the truth values for a subset of those, and the task is to find the most likely assignment to all other atoms, that is, the possible world with the highest probability in which the evidence holds. As in [16], we assume ProbLog programs with finite groundings[2].

For a set of **annotated disjunctions** $A = \{a_1, .., a_k\}$, the most probable explanation boils down to finding the selection in $Tr(A)$ with highest probability, amongst the ones in which the evidence holds. Given that a selection defines an interpretation, the MPE state is the truth value assignments according to this interpretation. Let $\mathcal{S}^{E=e} = \{\sigma | I(\sigma) \models E = e\}$ be the set of selections in which the evidence holds and $\hat{\sigma} = argmax_{\sigma \in \mathcal{S}^{E=e}} P(\sigma)$, then $MPE_A(E) = I(\hat{\sigma})$.

[2]This is known as the finite support assumption, and can be achieved by considering the relevant ground program (see Chapter 2).

Example 5.6. *For the set of ADs in Example 5.3, the evidence* `blue(b1)` *= false makes* σ_3 *invalid. The MPE state given the evidence is the one selection among* σ_1, σ_2 *and* σ_4 *with the highest probability. This is* $\sigma_4 = \{$`no_pick(b1)`$\}$ *with probability* 0.4. *That is, in the MPE state* `no_pick(b1)` *is true.*

The ProbLog encoding of these ADs (cf. Example 5.5) has 8 *possible worlds defined by the* 3 *probabilistic facts of the encoding. These possible worlds and their probabilities are (*`p(b1)` *and* `np(b1)` *are short for* `pick(b1)` *and* `no_pick(b1)`*, respectively):*

Possible World	`pf(1, 1)`	`pf(1, 2)`	`pf(2, 1)`	`red(b1)`	`green(b1)`	`blue(b1)`	`p(b1)`	`np(b1)`	$P(\omega_i)$
ω_1	T	T	T	T	F	F	T	F	*0.27*
ω_2	T	T	F	F	F	F	F	T	*0.18*
ω_3	T	F	T	T	F	F	T	F	*0.09*
ω_4	T	F	F	F	F	F	F	T	*0.06*
ω_5	F	T	T	F	T	F	T	F	*0.18*
ω_6	F	T	F	F	F	F	F	T	*0.12*
ω_7	F	F	T	F	F	T	T	F	*0.06*
ω_8	F	F	F	F	F	F	F	T	*0.04*

The evidence `blue(b1)` *= false holds in all worlds except* ω_7. *The MPE task, according to Definition 5.2, is to identify the possible world with the highest probability. Possible world* ω_1 *has the highest probability (*0.27*). Then, the MPE state of the ProbLog program is* {`pf(1, 1)` *= true,* `pf(1, 2)` *= true,* `pf(2, 1)` *= true*}. *For this MPE state* `no_pick(b1)` *is false and* `pick(b1)` *is true, which is different from the MPE state of the underlying set of ADs.* △

Example 5.6 illustrates that using the ProbLog encoding of annotated disjunctions can result in incorrect MPE states. The reason is that several possible worlds of the ProbLog encoding may correspond to the same selection in the probability tree – in Example 5.6 possible worlds $\omega_1, \omega_3, \omega_5$ and ω_7 correspond to one selection, namely to σ_4. This is not a problem when computing marginal probabilities, as the probabilities of all possible worlds where the query is true are summed in this case.

5.3.1 MPE as MAP

In [16] MPE is mentioned as a special case of the more general maximum a posteriori (MAP) inference. MAP is the task of finding the most likely values for a set of query atoms given *partial* evidence. The query atoms are a subset of all atoms of the ProbLog program for which evidence is not given. Solving then the MAP task requires to marginalize over the atoms which are neither given as queries nor as evidence. That is, we have to apply maximization over the query atoms given the evidence atoms and summation over the rest. MAP inference

can be used in order to find the MPE state of a ProbLog program with ADs. For the ProbLog encoding of such a program we can give as queries all the atoms of that program excluding the probabilistic facts generated by the encoding. Then solving the MAP task will give the truth value assignments of these atoms which is in practice the MPE state of the initial program. The MAP inference is computationally expensive, while MPE can be computed efficiently [26, 10]. The new encoding we introduce in Section 5.4.1 allows to solve the MPE task on ProbLog programs with ADs both correctly and efficiently by enforcing a one-to-one correspondence between selections and possible worlds.

5.3.2 Relation to Most Probable Proof

In PLP the term *most probable explanation* typically is used interchangeably with *most probable proof*, also called *Viterbi proof* [58, 67, 32, 3]. A proof (or explanation) ω' for a query is a partial truth value assignment (or partial possible world) such that for all full assignments extending the proof, the query holds. Finding a most probable proof (the VIT task) is different from MPE in that it does not aim at finding the state of all unobserved variables, but an assignment to a small set of variables sufficient to explain a query. More formally, given a query q, we have $VIT(q) = \arg\max_{\omega' \in E(q)} P(\omega')$ with $E(q)$ the set of all explanations or proofs of q.

For certain types of models, finding the Viterbi proof for query q corresponds to solving MPE with $q = true$ as evidence. This is true for instance in programs modeling Hidden Markov Models, where both VIT and MPE have to make one choice per time point, but does not hold in general:

Example 5.7. *In the program below, query* `win` *has two proofs: the first uses facts* `red` *and* `green` *and has probability* $0.4 \cdot 0.9 = 0.36$*, the second uses facts* `blue` *and* `yellow` *and has probability* $0.5 \cdot 0.6 = 0.3$*. Thus, the Viterbi proof for* `win` *is the first one. The MPE state for evidence* `win` $=$ *true, however, is* $\{\neg$`red`, `green`, `blue`, `yellow`$\}$*, as can be easily verified, and this state does not extend the Viterbi proof.*

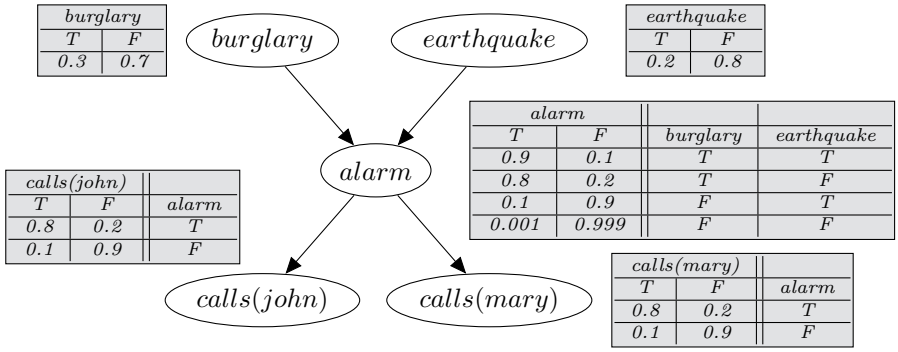
```
0.4::red.   0.9::green.          win :- red, green.
0.5::blue.   0.6::yellow.        win :- blue, yellow.
```

△

5.3.3 Relation to Most Probable Explanation for Bayesian Networks

To encode a Bayesian network (BN) using annotated disjunctions, each row in each conditional probability table (CPT) is encoded as a single AD to capture the value assignments and a the probability. There is only one CPT per variable and the rows in a CPT express mutually exclusive value assignments to a set of variables (the *parents* of a node). The parents of a node form the body of the AD.

Example 5.8. *Consider the well-known burglary-earthquake-alarm Bayesian network:*



Encoding the network as a ProbLog program with annotated disjunctions uses probabilistic facts the burglary *and the* earthquake *nodes. Probabilistic facts encode random events with binary outcome and therefore are suitable to represent the* burglary *and* earthquake *nodes of the network. With ADs we encode the other nodes. It is sufficient to use ADs with only one head atom (see Section 5.1.2). The body of an AD corresponds to the parent nodes with their value assignments.*

```
person(john).       person(mary).
0.3::burglary.      0.2::earthquake.

0.9::alarm <- burglary, earthquake.
0.8::alarm <- burglary, \+earthquake.
0.1::alarm <- \+burglary, earthquake.
0.001::alarm <- \+burglary, \+earthquake.

0.8::calls(X) <- alarm, person(X).
0.1::calls(X) <- \+alarm, person(X).
```

△

It has been shown that this encoding with ADs expresses the same probability distribution as the original Bayesian network [49]. We can shown that, given Definition 5.2, the MPE state of the Bayesian network is equivalent to the MPE state of a set of ADs that encode the Bayesian network.

The probability tree inferred by such a set of ADs has as property that each leaf has a unique interpretation. Therefore, the interpretation with the highest probability and the selection with the highest probability are equivalent. An intuitive proof can be constructed as follows: For any given CPT for a variable a, at each point in constructing the probability tree, if all parent variables of a certain CPT are present in the current partial interpretation, exactly one rule with a as head has a condition that is true. For each value of a a subtree is instantiated. None of the other rules can have a true condition in any of the subtrees due to the mutual exclusiveness. This guarantees that each subtree has a different value assigned to a and no two interpretations in different subtrees can be identical. When each leaf represents a unique interpretation, each selection maps to a unique interpretation with equal probability.

5.4 Weighted CNF Encoding for Annotated Disjunctions

The inference pipeline of ProbLog2 is based on a transformation of the initial ProbLog program to a Boolean formula in CNF that is next compiled into an sd-DNNF. The sd-DNNF allows efficient weighted model counting (WMC) in order to compute probabilities (see Chapter 2). We now introduce an encoding of annotated disjunctions in line with this reduction which uses ProbLog constraints (see Section 4.1) to retain the semantics of the ADs.

5.4.1 Encoding

The encoding of annotated disjunctions we propose has two parts: (i) a logic program with weighted facts that is transformed to CNF according to the grounding and Boolean formula conversion components of the ProbLog2 pipeline; and (ii) a set of constraints that is directly added to the CNF. This is a special case of cProbLog (see Section 4.1), adapted directly to the specific constraints needed here. ProbLog employs weighted model counting (WMC) techniques for MARG and MPE inference on the sd-DNNF that results from compiling the CNF.

Definition 5.3 (Weighted CNF encoding for ADs)**.** *The weighted CNF encoding (or wCNF encoding, in short) of a ground AD* $p_1 :: h_1; ..; p_n :: h_n \leftarrow b_1, .., b_m.$ *with unique identifier* $\mathtt{a_j}$ *consists of:*

- *for each* h_i *with* $1 \leq i \leq n$ *a* surrogate probabilistic fact $\mathtt{spf(a_j, h_i, i)}$ *with* $\mathit{true}(\mathtt{spf(a_j, h_i, i)}) = p_i$ *and* $\mathit{false}(\mathtt{spf(a_j, h_i, i)}) = 1.0$ *and a clause* $\mathtt{h_i} : -\mathtt{b_1}, .., \mathtt{b_m}, \mathtt{spf(a_j, h_i, i)}.$
- *(the conjunction of) the following two constraints:*

$$\bigwedge_{i=1}^{n-1} \bigwedge_{l=i+1}^{n} (\neg \mathtt{spf(a_j, h_i, i)} \vee \neg \mathtt{spf(a_j, h_l, l)}), \qquad \bigwedge_{k=1}^{m} \mathtt{b_k} \Leftrightarrow \bigvee_{i=1}^{n} \mathtt{spf(a_j, h_i, i)}$$

Intuitively, surrogate probabilistic facts make the choices in ADs explicit, and constraints ensure that this does not introduce undesired combinations of values. The first constraint ensures that at most one surrogate probabilistic fact for a given AD can be true at a time. The second constraint ensures that one surrogate probabilistic fact for a given AD will be true iff the body of the AD is true. When one head atom h_i is selected for an AD, the other head atoms are ignored. That is, they do not influence the probability of any selections with h_i true. That is why the false probability of a surrogate probabilistic fact is set to 1.0. The first constraint states that for each AD, at most one surrogate probabilistic fact can be true in any possible world, and thus only one head atom can be made true by the AD. The second constraint states that a choice is made if and only if the body of the AD is true. We write a surrogate probabilistic fact $\mathtt{spf(a_j, h_i, i)}$ with $\mathit{true}(\mathtt{spf(a_j, h_i, i)}) = p$ as $(p, 1.0) :: \mathtt{spf(a_j, h_i, i)}$. .

Example 5.9. *The wCNF encoding of the two ADs of Example 5.3 consists of the program part:*

```
(0.6, 1.0)::spf(1, red(b1), 1).
(0.3, 1.0)::spf(1, green(b1), 2).
(0.1, 1.0)::spf(1, blue(b1), 3).
red(b1):- pick(b1), spf(1, red(b1), 1).
green(b1):- pick(b1), spf(1, green(b1), 2).
blue(b1):- pick(b1), spf(1, blue(b1), 3).
(0.6, 1.0)::spf(2, pick(b1), 1).
(0.4, 1.0)::spf(2, no_pick(b1), 2).
pick(b1):- spf(2, pick(b1), 1).
no_pick(b1):- spf(2, no_pick(b1), 2).
```

and the four constraints where in the last one we omit the equivalence with true:

$$(\neg \texttt{spf}(1, \texttt{red}(\texttt{b1}), 1) \vee \neg \texttt{spf}(1, \texttt{green}(\texttt{b1}), 2)) \wedge$$

$$(\neg \texttt{spf}(1, \texttt{red}(\texttt{b1}), 1) \vee \neg \texttt{spf}(1, \texttt{blue}(\texttt{b1}), 3)) \wedge$$

$$(\neg \texttt{spf}(1, \texttt{green}(\texttt{b1}), 2) \vee \neg \texttt{spf}(1, \texttt{blue}(\texttt{b1}), 3))$$

$$\texttt{pick}(\texttt{b1}) \Leftrightarrow (\texttt{spf}(1, \texttt{red}(\texttt{b1}), 1) \vee \texttt{spf}(1, \texttt{green}(\texttt{b1}), 2) \vee \texttt{spf}(1, \texttt{blue}(\texttt{b1}), 3))$$

$$\neg \texttt{spf}(2, \texttt{pick}(\texttt{b1}), 1) \vee \neg \texttt{spf}(2, \texttt{no\_pick}(\texttt{b1}), 2)$$

$$\texttt{spf}(2, \texttt{pick}(\texttt{b1}), 1) \vee \texttt{spf}(2, \texttt{no\_pick}(\texttt{b1}), 2)$$

The possible worlds of the program in which the constraints hold are (`r`*,* `g`*,* `b`*,* `p`*,* `np` *abbreviate* `red(b1)`*,* `green(b1)`*,* `blue(b1)`*,* `pick(b1)`*,* `no_pick(b1)`*):*

Possible World	`spf(1,r,1)`	`spf(1,g,2)`	`spf(1,b,3)`	`spf(2,p,1)`	`spf(2,np,2)`	`r`	`g`	`b`	`p`	`np`	$P(\omega_i)$
ω_1	T	F	F	T	F	T	F	F	T	F	*0.36*
ω_2	F	T	F	T	F	F	T	F	T	F	*0.18*
ω_3	F	F	T	T	F	F	F	T	T	F	*0.06*
ω_4	F	F	F	F	T	F	F	F	F	T	*0.40*

△

Comparing Example 5.9 to Example 5.5 and Example 5.3 shows that our encoding results in a set of possible worlds that (i) have the same truth value assignments and (ii) the same probabilities. In contrast the set of possible worlds in Example 5.5 is inconsistent with the semantics of ADs. That is, we can compute the correct MPE state from the set of possible worlds listed in the table of Example 5.9. Furthermore, as there is a one-to-one correspondence between the possible worlds and selections we can perform all other inference tasks correctly.

In the next section we formally prove correctness for our approach.

5.4.2 Correctness

We prove correctness of our encoding for probabilistic inference in two steps. First, we show that for a set of annotated disjunctions, there is a one-to-one mapping between the models of the wCNF and the selections in the probability

tree (cf. Section 5.1.2), and second, that the weight of a model of the CNF is the probability of the corresponding selection.

Theorem 5.1. *For a set $A = \{a_1, \ldots, a_k\}$ of ground annotated disjunctions, there is a bijection from the set $\mathcal{M}$ of models of the wCNF for A to the set $\mathcal{S}$ of selections in a probability tree Tr for A.*

***Proof**: Let L_A be the set of atoms in A, and L_F the set of surrogate facts in the wCNF encoding of A. For every truth value assignment l_F to L_F, there is exactly one truth value assignment l_A to L_A such that $l_F \cup l_A$ is a model of the program part of the encoding [16]. The first constraint filters out all assignments l_F that assign true to more than one surrogate fact for the same ground AD, and the second filters out those that assign true to any surrogate fact for an AD whose body is false in $l_F \cup l_A$. Each remaining assignment $l_F \cup l_A$ is in one-to-one correspondence with a selection in $\mathcal{S}$. That is, each such an assignment (i.e., a model) corresponds to exactly one path from the root to a leaf of the tree Tr (the assignment l_A is a model of the node's interpretation) and there is one model for each path.* ■

Theorem 5.2. *Given a set $A = \{a_1, \ldots, a_k\}$ of ground annotated disjunctions, the set $\mathcal{M}$ of models of the wCNF for A, and the set $\mathcal{S}$ of selections in a probability tree Tr for A, the weight of a model $M \in \mathcal{M}$ equals the probability of the corresponding selection $S \in \mathcal{S}$.*

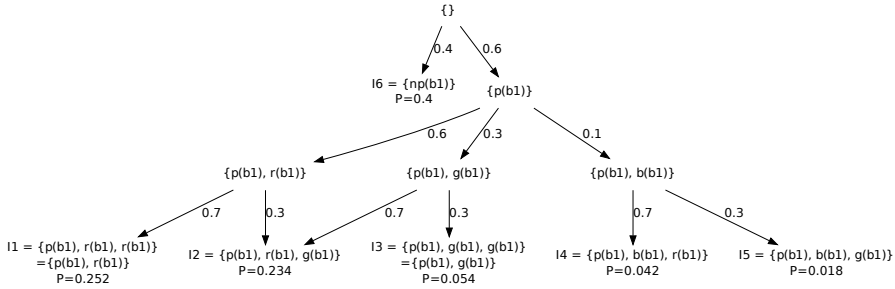
***Proof**: Follows directly from the fact that the model and the selection follow the same path through the tree and the definition of the weight function on the CNF. At each node, the probability of the selection so far is multiplied with the probability p_i of the chosen head atom, and the weight of the model with the weight of the AD's surrogate facts, $p_i \cdot 1.0 \cdot \ldots \cdot 1.0 = p_i$.* ■

The validity of Theorem 5.1 and Theorem 5.2 can be verified by comparing the possible worlds of the wCNF in Example 5.9 with the selections associated with the probability tree in Example 5.3: (i) each possible world is associated with exactly one selection and vice-versa (consistent with Theorem 5.1) and (ii) the probability of each possible world equals the probability of the selection that is in bijection with this possible world (consistent with Theorem 5.2). Later, in Example 5.10 we observe the correctness of our encoding (with respect to Theorem 5.1 and Theorem 5.2) for multiple ADs that have the same atoms in their heads.

5.4.3 Annotated disjunctions and multiple causes

Example 5.10 illustrates how our approach encodes a more special case where multiple ADs have the same atoms in their heads. That is, the same event can result from multiple causes.

Example 5.10. *Consider again the same problem as in the previous examples: a bag with colorful balls. According to one source of information, in the bag there are red, green and blue balls (as in Example 5.3), while another source states that there are only red and green balls in the bag:*
r_1*:* `0.6::red(b1); 0.3::green(b1); 0.1::blue(b1) <- pick(b1).`
r_2*:* `0.7::red(b1); 0.3::green(b1) <- pick(b1).`
r_3*:* `0.6::pick(b1); 0.4::no_pick(b1) <- true.`
The probability tree associated with these ADs is:



and the selections corresponding to the paths from the root to a leaf are:

Selection:	*Interpretation*	*Complete*	*Probability* $P(\sigma_i)$
σ_1	$I1 = \{$`pick(b1)`, `red(b1)`$\}$	✓	0.252
σ_2	$I2 = \{$`pick(b1)`, `red(b1)`, `green(b1)`$\}$	✓	0.108
σ_3	$I2 = \{$`pick(b1)`, `red(b1)`, `green(b1)`$\}$	✓	0.126
σ_4	$I3 = \{$`pick(b1)`, `green(b1)`$\}$	✓	0.054
σ_5	$I4 = \{$`pick(b1)`, `red(b1)`, `blue(b1)`$\}$	✓	0.042
σ_6	$I5 = \{$`pick(b1)`, `green(b1)`, `blue(b1)`$\}$	✓	0.018
σ_7	$I6 = \{$`no_pick(b1)`$\}$	×	0.4

In this case the interpretation $I2$ is defined by two selections (σ_2 and σ_3). According to the definition, the MPE state is the interpretation associated with the selection with highest probability (in this case thus $I1$). Although the two

selections σ_2 and σ_3 define the same interpretation when computing the MPE σ_2 and σ_3 need to be considered apart.

The following table states the possible worlds corresponding to the wCNF encoding of this program:

Poss. World	*spf(1,r,1)*	*spf(1,g,2)*	*spf(1,b,3)*	*spf(2,r,1)*	*spf(2,g,2)*	*spf(3,p,1)*	*spf(3,np,2)*	r	g	b	p	np	$P(\omega_i)$
ω_1	T	F	F	T	F	T	F	T	F	F	T	F	0.252
ω_2	F	T	F	T	F	T	F	T	T	F	T	F	0.126
ω_3	F	F	T	T	F	T	F	T	F	T	T	F	0.042
ω_4	T	F	F	F	T	T	F	T	T	F	T	F	0.108
ω_5	F	T	F	F	T	T	F	F	T	F	T	F	0.054
ω_6	F	F	T	F	T	T	F	F	T	T	T	F	0.018
ω_7	F	F	F	F	F	F	T	F	F	F	F	T	0.4

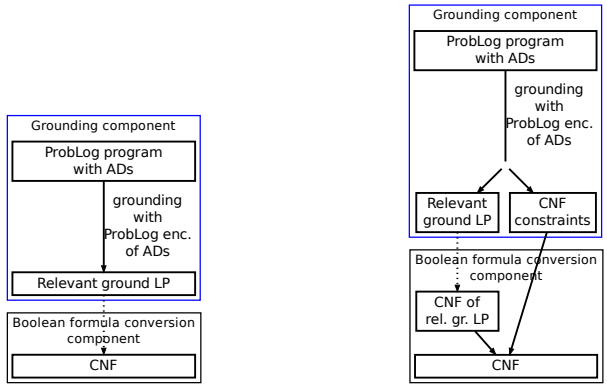
△

5.5 Implementing the wCNF AD encoding and the MPE task in a ProbLog pipeline

The inference pipeline of a ProbLog system consists of four main components: *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation* as presented in Chapter 2. Each previous component produces input for the next one and it may consist of one or more processing steps. ADs are processed during grounding – when an AD needs to be proven the grounder invokes a processing step to generate the necessary probabilistic facts and Prolog rules (in the case of the ProbLog encoding) or the set of surrogate probabilistic facts, Prolog rules and constraints (in the case of the wCNF encoding).

We implemented our approach in a ProbLog2 pipeline, that is, using grounding to a relevant ground LP, Boolean formula conversion to a CNF and Knowledge compilation with c2d or DSHARP to an sd-DNNF.

The result of grounding a ProbLog program with ADs using the ProbLog encoding is the relevant ground LP; using the wCNF encoding, on the other hand, generates (i) a relevant ground LP and (ii) a CNF of the constraints to preserve the semantics of the ADs. In the second component the relevant ground LP is converted to a Boolean formula in CNF. For the wCNF encoding the CNF conversion needs to be conjoined with the CNF of the constraints generated by the encoding. This CNF is used in the knowledge compilation component to generate an sd-DNNF, which is then evaluated with respect to the inference task. Figure 5.1 illustrates the differences between the ProbLog encoding and the wCNF encoding with respect to their implementation in the ProbLog2 pipeline.



a. ProbLog encoding b. Weighted CNF encoding

Figure 5.1: Differences between the implementation of the ProbLog and the wCNF encodings in the ProbLog2 pipeline.

For the MARG and COND tasks the sd-DNNF is converted to an arithmetic circuit (AC) by substituting logical operators with mathematical – conjunction is substituted with multiplication and disjunction with summation. For MPE inference we use the approach in [10, Chapter 12]. That is, in order to solve the MPE task we need to substitute disjunction with the operation of *maximum*, so that when we traverse the AC we can determine the one possible world with the maximum probability. Another difference with evaluation for MARG or COND inference is that the result of the MPE task is the MPE state, rather than the probability of a query. That requires to keep track of the atoms and their truth values that define the possible world with the maximum probability. Hence, when traversing the AC we accumulate the atoms and their truth values in a set that forms the MPE state. Given the wCNF encoding the result of this evaluation will contain surrogate probabilistic facts together with their truth values. The surrogate probabilistic facts are part of the encoding and we do not present them to the user, rather we perform a postprocessing of the result to extract the head atoms of the ADs as given in the initial ProbLog program with ADs.

5.6 Evaluation

In this section we present theoretical and empirical evaluation of the wCNF encoding for annotated disjunctions. First, we give a theoretical analysis of the complexity of our encoding. Then we perform a series of experiments in which

Encoding:	Number of CNF Variables:	Number of CNF Clauses:
ProbLog:	$2n + m - 1$	$\frac{n(2m+n+3)}{2} - 1$
wCNF:	$2n + m$	$\frac{n(4m+n+3)}{2} + 1$

Table 5.1: Size of generated CNFs by the ProbLog and the wCNF encoding.

we compare the wCNF to the ProbLog encodings. We compare the performance of the two approaches on the MARG tasks, since it is one of the inferences tasks for which the ProbLog encoding is also correct. We aim to determine whether the new encoding is better and should substitute the ProbLog encoding.

5.6.1 Analysis

As shown in Figure 5.1 b), using the wCNF encoding will lead to two different CNF formulae – (i) the Boolean formula that results from applying the Boolean formula conversion on the relevant ground LP φ_r and (ii) the CNF formula of the constraints that are generated by the encoding φ_{ADs}. In case the inference task is COND or MPE there is also the set of evidence that needs to be considered in building the CNF, that is, the final CNF contains a conjunction of the evidence atoms φ_E that states that the evidence holds. The resulting CNF, which is given as input to the knowledge compilation component is $\varphi = \varphi_r \wedge \varphi_{ADs} \wedge \varphi_E$. For an AD with $|head| = n$ and $|body| = m$, the corresponding ProbLog encoding has n rules, with the n^{th} rule containing $m + n - 1$ facts in its body. The wCNF encoding constructs rules with constant body size $(m + 1)$. We assume that no AD introduces a cycle. Such assumption does not alter our analysis. The reason is that the proof-based cycle handling approach (see Chapter 2) that we use in the Boolean formula conversion has the same impact on both encodings.

After the cycle handling we use the approach discussed in Chapter 3, Section 3.4.4 to generate the CNF φ. We compute the number of clauses and variables in φ that results from the different encodings of the AD. The results are listed in Table 5.1.

Table 5.1 shows that the ProbLog encoding produces smaller CNFs as compared to the wCNF Encoding. A head atom may appear in different ADs. Both encodings deal with each AD independently and define one (ground ProbLog) rule for each dependency between a head atom and the probabilistic facts generated by the encoding. The bodies of rules with the same head are combined in a disjunction in the CNF and introduce additional variables and clauses. For the two encodings of a set of ADs which share head atoms the number of additional variables and clauses in the CNF is the same. Table 5.1 ignores the

```
win(P):-red(P, 0, 0).
win(P):-green(P, 0, 0).
win(P):-blue(P, 0, 0).

0.3::red(0, SG, SB); 0.3::green(0, SG, SB); 0.4::blue(0, SG, SB) <- true.

0.3::red(T, SG, SB); 0.3::green(T, SG, SB); 0.4::blue(T, SG, SB) <-
    T > 0, SG < 2, SB < 3, Tprev is T - 1, red(Tprev, 0, 0).
0.3::red(T, SG, SB); 0.3::green(T, SG, SB); 0.4::blue(T, SG, SB) <-
    T > 0, SG < 2, SB < 3, Tprev is T - 1, SGNew is SG + 1, green(Tprev, SGNew, SB).
0.3::red(T, SG, SB); 0.3::green(T, SG, SB); 0.4::blue(T, SG, SB) <-
    T > 0, SG < 2, SB < 3, SBNew is SB + 1, Tprev is T - 1, blue(Tprev, SG, SBNew).
```

Figure 5.2: The *Balls* ProbLog program.

additional clauses and variables in the cases where the ADs share head atoms and the ones introduced during cycle handling because they are the same for both encodings.

For knowledge compilation to sd-DNNF ProbLog2 uses either c2d [9] or DSHARP [54]. These compilers are non-deterministic (see [8] for more details), that is, for the same CNF the resulting sd-DNNFs may differ. Hence, we cannot make theoretical estimations on the time for generating an sd-DNNF, its size or the time for its evaluation.

Our implementation of the evaluation component for the MPE task is according to [10, Chapter 12] and has the same complexity as the evaluation for the MARG task. Our experiments confirmed this statement. Details on the complexity of ProbLog's inference can be found in [14].

5.7 Experimental Data

Our experiments aim to answer: **(i) What is the trade-off between the ProbLog encoding and the wCNF encoding when performing MARG inference?**; **(ii) How does the wCNF encoding scale w.r.t. the data size?**.

We analyze the time and memory consumption from experimenting on three artificially generated datasets. The first one (the *Balls* benchmark) is a more complex version of the ball game (cf. Example 5.3) that uses ADs to represent the different possible consequences of an action. We used this benchmark set in our experiments in Chapter 2 and Chapter 3.

The two other benchmarks are taken from [50]: we use the programs with annotated disjunctions with increasing head atoms (M_{gh}) and the ones with

```
0.47::a0 <- a1.
0.96::a1 <- true.
0.31::a0; 0.69::a1 <- a2.
0.41::a2 <- true.
0.04::a0; 0.14::a1; 0.82::a2 <- a3.
0.52::a3 <- true.
```

a. M_{gh} dataset.

```
0.91::a0 <-    a1.
0.70::a0 <- \+ a1,    a2.
0.36::a0 <- \+ a1, \+ a2,  a3.
0.59::a1 <-    a2.
0.17::a1 <- \+ a2,  a3.
0.79::a2 <-    a3.
0.46::a3.
```

b. M_{gnb} dataset.

Figure 5.3: Examples from the M_{gh} and M_{gnb} datasets with 4 variables.

increasing number of negated body atoms (M_{gnb}). With the M_{gh} dataset we show the impact of the number of heads on the performance of the encodings. The M_{gnb} dataset shows the impact of the size of the bodies (that is, the second constraint of the wCNF encoding).

Our benchmarks can be found at:
`people.cs.kuleuven.be/~dimitar.shterionov/mpe`.

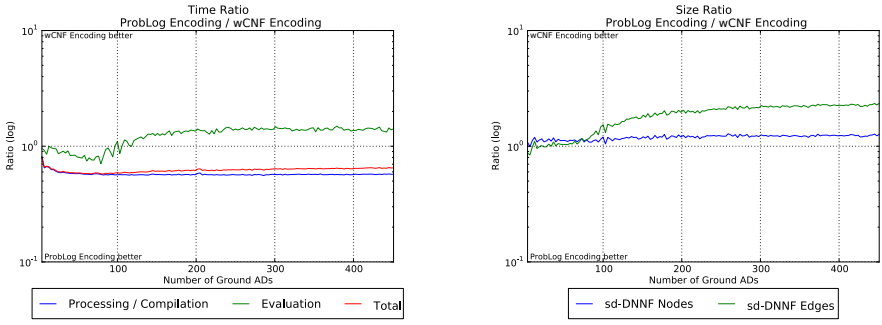
To assess the performance of our method we measure the *processing-compilation* time (time for grounding plus Boolean formula conversion plus knowledge compilation), the evaluation time and the total inference time. We report the ratio $T^r = \frac{T(ProbLog)}{T(wCNF)}$ for programs of incremental size which shows the relative time between the ProbLog and the wCNF encoding. To assess the memory consumption we compare the number of nodes and edges for the generated sd-DNNF in each test run. We report the size ratio $S^r = \frac{S(ProbLog)}{S(wCNF)}$.

We use the c2d compiler in our experiments. Since this compiler may generate different sd-DNNFs for the same CNF we run each test 5 times and report the average of time and size. We run our experiments on an 8-thread, Intel®Core™i7 @ 3400 MHz machine with 16GB RAM.

5.8 Experimental Results

Figure 5.4 summarizes the results for the *Balls* program. Figure 5.4.a shows the time ratio T^r w.r.t. the number of ADs and Figure 5.4.b the size ratio S^r.

Figure 5.4.a shows a worse processing-computation time (blue line) for the wCNF encoding in comparison to the ProbLog encoding but better evaluation time (green line). Since the process-compilation time is much larger than the evaluation, the total time (red line) is also worse for the wCNF encoding. Furthermore, the total time for the case of the wCNF encoding is higher (as



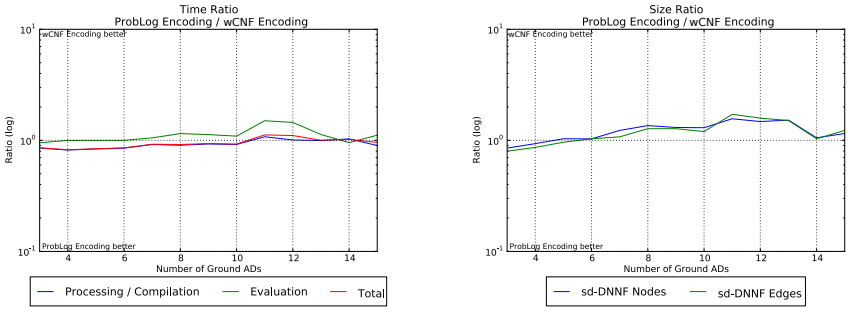
a. b.

Figure 5.4: Results from the *Balls* benchmark.

compared to the ProbLog encoding) with a constant factor over the number of ground ADs.

From Figure 5.4.b we conclude that for almost all queries the compiled sd-DNNFs for the wCNF encoding are smaller. This, together with the better evaluation time shows the wCNF encoding to be preferable, since for many real-world problems the model is compiled once and evaluated many times, e.g., diagnosis or prognosis.

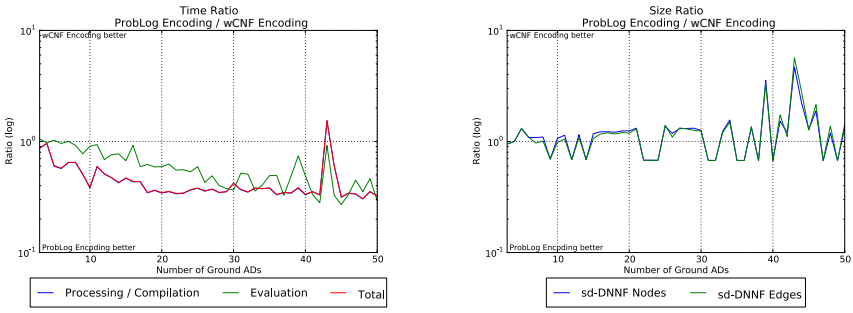
The results from experimenting on the M_{gh} dataset, summarized in Figure 5.5, show similar tendencies as the ones for the *Balls* benchmark. That is, the wCNF encoding has worse processing-compilation and therefore also total execution time but still better evaluation time.



a. b.

Figure 5.5: Results from the M_{gh} benchmark.

Crucial for the wCNF encoding is the size of the body of the AD as it influences drastically the size of the CNF (cf. Table 5.1). Figure 5.6 shows how this influences the execution time, that is, for the M_{gnb} dataset the wCNF encoding results in worse processing-compilation, total and also evaluation time compared to the ProbLog encoding.



a. b.

Figure 5.6: Results from the M_{gnb} benchmark.

Analyzing the results of our experiments we can state that there is a trade-off between the ProbLog encoding and the wCNF encoding: generally, the total execution time for inference is worse for the wCNF encoding case. But the better evaluation time (*Balls* and M_{gh} benchmarks) due to the more compact sd-DNNFs makes our encoding preferable for problems where you compile once and evaluate many times, such as learning and diagnosis. Also, in extreme cases, like the M_{gnb} benchmark, the wCNF encoding does not perform better.

Our experiments also show that with the two encodings ProbLog's inference scales in similar ways, with the ProbLog encoding outperforming the wCNF encoding with respect to total execution time.

5.9 Conclusions

Motivated to provide support for the MPE inference task on ProbLog programs with annotated disjunctions (ADs) we developed a new encoding of ADs. ADs were supported previously by transforming them into probabilistic facts and Prolog rules, i.e., into a ProbLog program in the core ProbLog language. This approach (the *ProbLog encoding*) was correct for the MARG (and COND) task but not for the MPE task. Our new encoding, i.e., the *wCNF Encoding*, also converts ADs into ProbLog facts and rules, but in addition it uses a form of

cProbLog constraints to ensure semantic equivalence between the transformed program and the original ProbLog program with ADs.

In this chapter we introduced the wCNF encoding, we proved correctness both for the MARG (and COND) and the MPE tasks. We then compared its performance to the ProbLog encoding with respect to MARG inference. Our experiments showed that our new encoding has to be used for MPE inference, because of its correctness for this inference task, and is preferable for MARG inference for problems that require more than one evaluation (as it is the case for e.g. diagnostics problems). For the typical MARG task, the ProbLog encoding is more efficient.

We also implemented the MPE inference tasks for ProbLog. We adapting the evaluation component of a ProbLog2 pipeline so that the MPE state can be correctly computed from an arithmetic circuit.

Chapter 6

Conclusion and Future Work

Probabilistic inference refers to a set of computational tasks used to derive new knowledge from probabilistic data. These tasks are in general computationally expensive and require efficient techniques in order to solve real-world problems. This thesis focused on the analysis, design and implementation of probabilistic inference pipelines – an architecture template employed by the Probabilistic Logic Programming (PLP) framework ProbLog to solve efficiently numerous inference tasks. The ProbLog framework consists of the ProbLog language and an inference pipeline. An inference pipeline is a sequence of transformation steps, called components, that reduce the computationally expensive inference task to a simple weighted model counting (WMC) problem. The inference pipeline of ProbLog employs four main components – grounding, Boolean formula conversion, knowledge compilation and evaluation. Each component can be implemented by different tools or algorithms. We presented a thorough analysis of the existing implementations of each component and investigated their performance. We considered two grounding approaches – grounding to nested tries used in ProbLog1 and MetaProbLog, and grounding to relevant ground logic program employed by ProbLog2; two approaches to convert cyclic groundings to Boolean formulae, namely, the proof-based and the rule-based approaches; three knowledge compilation techniques – knowledge compilation to Reduced Ordered Binary Decision Diagrams (ROBDDs) with the compiler SimpleCUDD, knowledge compilation to Smooth Deterministic Decomposable Negation Normal Form (sd-DNNF) with the c2d compiler or with the Dsharp compiler; one evaluation method for ROBDDs and two evaluation methods for sd-DNNFs. From all possible combinations we excluded pipelines with theoretically high complexity and focused on 14 pipelines. Then we evaluated these pipelines on a large set of benchmarks. Our results showed that while the

knowledge compilation component is the component with the highest complexity it is not its implementation that has a crucial impact on the overall system performance. Rather it is the type and the complexity of the formula generated by the Boolean formula conversion component that strongly affect knowledge compilation. In particular, we showed that feeding a BDD script that was generated from a Boolean formula in CNF to the SimpleCUDD compiler (i) increases the knowledge compilation time and (ii) produces complex ROBDDs, inefficient to evaluate. Moreover, one of the newly created pipelines that combines grounding to relevant ground LP and knowledge compilation to ROBDDs via a translation of the Boolean formula into a BDD script that does not involve rewriting a CNF, outperforms the rest on many of our benchmarks.

In order to optimize the Boolean formula we devised a method that detects frequent subfromulae patterns and uses them to rewrite the formula into a more compact representation. Empirical evaluation showed that our compaction method reduces the run time for knowledge compilation to ROBDDs and to sd-DNNFs with c2d; and increases the run time for knowledge compilation to sd-DNNFs with Dsharp. These results implied that optimizing Boolean formulae should not always aim at the most compact form, but rather consider the type of application and the tool that will be used for knowledge compilation. We then supported this statement by performing a more relaxed compaction that reduces the number of clauses but does not necessarily reduce the number of Boolean variables in the formula. The results showed that knowledge compilation with Dsharp has improved significantly.

With that said, we ought to note the recent advances in knowledge compilation and their applicability in ProbLog. The latest release of the ProbLog system employs a new knowledge compilation approach. Namely, knowledge compilation to Sentential Decision Diagrams (SDDs) [11]. While it shows promising results for ProbLog inference [89] it is susceptible to the same problems of the input formula representation as in the case of compiling a CNF into ROBDDs discussed in Chapter 2. That is why in the future we aim at further improving the input Boolean formula conversion by employing a smarter technique that depends on the knowledge compilation method. For this purpose we plan to investigate a method to classify ProbLog programs and determine the optimal settings of ProbLog inference pipelines.

In addition to the research on the pipeline architecture of ProbLog in this thesis we presented two extensions of the ProbLog language: (i) First-Order Logic (FOL) constraints and (ii) Annotated Disjunctions (ADs). Incorporating constraints into ProbLog gave rise to the language cProbLog. cProbLog increases the expressive power of ProbLog. Its semantics generalizes evidence to FOL sentences. We devised the first implementation for inference with cProbLog programs based on this generalization property, that converts constraints to

ProbLog programs with evidence. In order to speed-up our method, we presented an optimization technique that reduces the size of the grounding of constraints and therefore, improves the inference performance.

Even though annotated disjunctions had already been incorporated in ProbLog, the existing encoding of ADs was correct for some of the inference tasks and incorrect for others. We proposed a new encoding of ADs by means of constraints that retains the semantics of ADs and as such, is correct for all inference tasks. We then implemented the most probable explanation task for ProbLog programs with ADs.

In the future we look upon two main research directions. One is related to improving the inference performance of ProbLog, and in general probabilistic logic programming software, by (i) optimizing the components and their input/output dependencies and (ii) implementing an intelligent inference pipeline, that selects its optimal parameters given the initial problem and the inference task. The other direction is applications. ProbLog is a general PLP language and it can provide solutions for many scientific communities like marketing, health care, natural language processing and others. Providing general methodologies to model and reason with problems from these domains would bridge the gap between PLP and these scientific fields.

Appendix A

Knowledge Compilation and Evaluation of Arithmetic Circuits

In this appendix we give more insights on the two target compilation languages that we use for ProbLog inference, namely sd-DNNFs and ROBDDs. We use the following Boolean formula in CNF as a running example.

$$a \wedge (b \vee c)$$

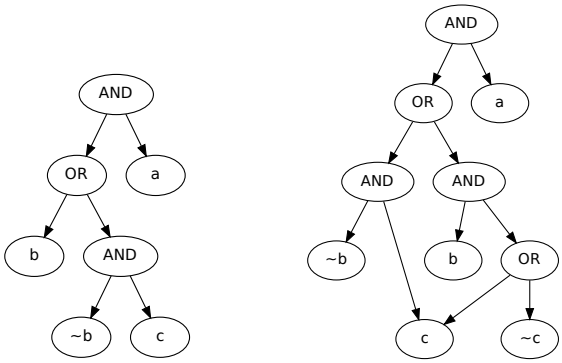
We assign the following probabilities to each atom: $0.6 - a$, $0.7 - b$ and $0.8 - c$.

A.1 sd-DNNFs

[12] presents a classification of knowledge compilation target languages, among which are the sd-DNNFs and ROBDDs. Both of them are members of the family of the Negation Normal Form languages. A formula in Negation Normal Form (NNF) is a rooted directed acyclic graph (DAG) where each leaf node is labeled with *true*, *false*, a variable or it's negation. The inner nodes are labeled with conjunctions or disjunctions. When the children of every conjunction (AND) node do not share any variables then the NNF has the property of Decomposability (DNNF); if the children of each disjunction node (OR) are logically contradictory then the NNF has the property of determinism (d-NNF);

if the children of each disjunction node, mention the same variables then the NNF has the property of Smoothness (s-NNF). The sd-DNNF language, which has all these properties, allows weighted model counting in linear to the tree width time.

For our example the d-DNNF and the sd-DNNF are given in Figure A.1.



a. d-DNNF. b. smooth d-DNNF.

Figure A.1: A d-DNNF encoding our example Boolean formula.

To compute the weighted model count of a formula represented as an sd-DNNF first the sd-DNNF should be interpreted as an arithmetic circuit (AC). Each leaf node representing a literal x is replaced by an AND node with two children – one that contains the indicator variable $\lambda[x]$ and another one that contains the weight variable $\theta[x]$; then each logical operator is substituted with a mathematical one – AND ($\wedge$) is substituted by multiplication ($*$) and OR ($\vee$) by summation ($+$). Figure A.2 shows the AC extracted from the sd-DNNF of Figure A.1 b).

To evaluate an AC we use the algorithm described in [10]. The algorithm traverses the AC twice: (i) a bottom-up traversal to compute the value of the AC; and (ii) a top-down traversal to compute the derivative with respect to a particular variable. These are the same computations that we perform on the program function, Chapter 2, Section 2.1.4, in order to compute the probability of a query.

Here we present our implementation of the aforementioned algorithm – i.e. the depth-first evaluation of sd-DNNFs discussed in Chapter 2, Section 2.3). Procedure 1 performs the bottom-up traversal to compute the value of an arithmetic circuit; Procedure 2 computes the derivative of a literal of interest,

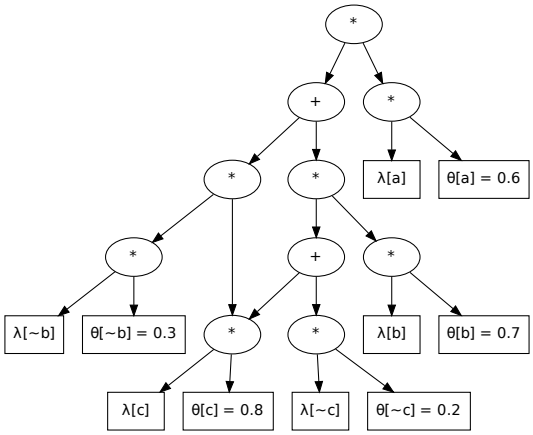


Figure A.2: An arithmetic circuit.

e.g., a query. We require the following prior initializations: (i) all λ nodes are given the value 1.0; (ii) each $\theta[x]$ node is given as value the probability $p(x)$ and each $\theta[\sim x]$ the value $(1 - p(x))$; (iii) all non leaf nodes are assigned a value *null* and (iv) the derivative of each non-root node is initialized with 0.0 and of the root it is initialized with 1.0.

Using Procedure 1 on the AC of Figure A.2 we compute its value: $((0.3 * 0.8) + ((0.8 + 0.2) * 0.7)) * 0.6 = 0.564$. The derivative of a node depends on the values of its parent nodes. To compute the derivative of c, i.e., of $\lambda[c]$[1] we need to consider the values of the two parents of the summation node that is a parent of $\lambda[c]$. In case the circuit is not smooth there will be only one parent of this node (see Figure A.1 a)) and the derivative will be incorrect with respect to the algorithm in Procedure 2. In particular, the derivative of $\lambda[c]$ is 0.6. If the circuit was constructed from non smooth d-DNNF then the result would be 0.18.

In order to use the DSHARP compiler [54] for ProbLog inference we implemented in addition the step to smooth the NNF. Our implementation is used in the distributable version of the tool for compiling to sd-DNNF.

[1]recall that the derivative is the measure of how a function changes when its input changes (Chapter 2, Section 2.1.4).

Procedure 1: COMPUTEVALUE(N)

Data: a node N (in the first call N is the root of an arithmetic circuit AC)
Result: the value v of N

```
if N.get_value() ≠ null then
    return N.get_value()
else
    C = N.get_children()
    if N.is_and() then
        v = 1.0
        for c ∈ C do
            v = v×computeValue(c)
        end
        N.set_value(v)
        return v
    else
        v = 0.0
        for c ∈ C do
            v = v+computeValue(c)
        end
        N.set_value(v)
        return v
    end
end
```

A.2 ROBDDs

The other target compilation language we use for ProbLog inference is the ROBDD language. It is a subset of the NNF language and has the properties of decomposability, decision and ordering. A decision node is a node labeled with $true$, $false$ or is an OR node of the form $(X \wedge \varphi) \vee (\neg X \wedge \psi)$, where X is a variable and φ, ψ are Boolean formula. The property of ordering is satisfied if for any two OR nodes N and M such that N is an ancestor of M $vars(N) < vars(M)$, where $vars(n_i)$ denotes all variables that appear in the path from the root to the node n_i and $<$ is an ordering function.

ROBDDs are most commonly represented as decision diagrams. In Figure A.3 we show an ROBDD in the NNF language and the corresponding Decision Diagram.

The algorithm to evaluate ROBDDs uses the decision diagram representation. This algorithm has been thoroughly described in [30, 41]. For completeness we outline it in Procedure 3. This method is used by the SimpleCUDD compiler

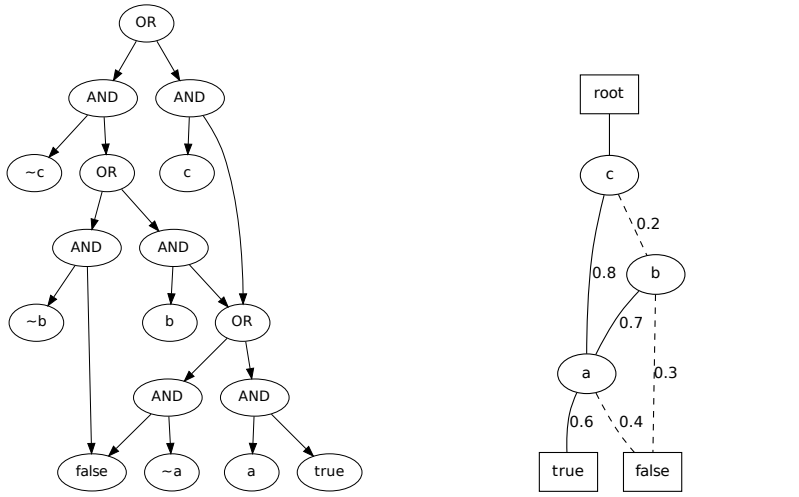
Procedure 2: computeDerivative(N)

Data: a node N (in the first call N is the root of an arithmetic circuit AC)
Result: the value v of N

```
if N.get_parents() ≠ ∅ then
    d = 0.0
    for p ∈ N.get_parents() do
        k = 1.0
        if p.type == * then
            if N.get_value() ≠ 0.0 then
                k = p.get_value() / N.get_value()
            else
                k = p.get_value()
            end
        end
        if p.get_derivative() > 0.0 then
            d = d + k × p.get_derivative()
        else
            d = d + k×computeDerivative(p)
        end
    end
    N.set_derivative(d)
    return d
else
    return null
end
```

to evaluate an ROBDD.

Procedure 3 computes the probability 0.564 for the ROBDD illustrated in Figure A.3 b).



a. ROBDD in the NNF language. b. ROBDD as a decision diagram.

Figure A.3: An ROBDD of our example Boolean formula.

Procedure 3: PROBABILITYROBDD(N).

Data: a node N
Result: the value v of N
if N *is true* **then**
  return 1.0
end
if N *is false* **then**
  return 0.0
end
if N.*accumulated_prob* $\neq$ *null* **then**
  return N.*accumulated_prob*
end
let h and l be the high and low children of N
let N.var_prob be the probability annotating the variable of node N
p_h = probabilityROBDD(h)
p_l = probabilityROBDD(l)
result = N.var_prob $\times$ p_h + (1 − N.var_prob) $\times$ p_l
N.accumulated_prob = *result*
return *result*

Appendix B

ProbLog Pipelines – Experimental Results

In this appendix we present the detailed results from our experiments with the 14 ProbLog pipelines discussed in Chapter 2.

B.0.1 Time Diagrams

MARG Inference

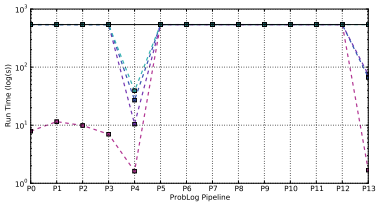


Figure B.1: Run times for the Alzheimer set, query $q1$.

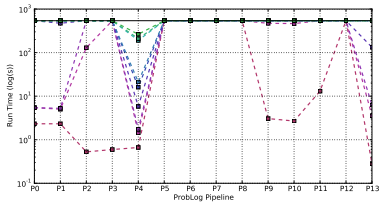


Figure B.2: Run times for the Alzheimer set, query $q2$.

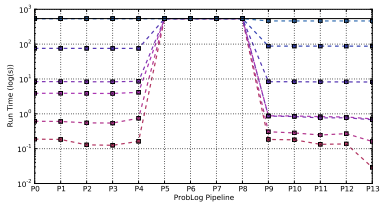


Figure B.3: Run times for the Alzheimer set, query $q3$.

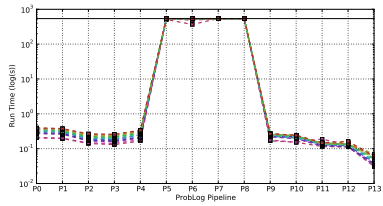


Figure B.4: Run times for the Alzheimer set, query $q4$.

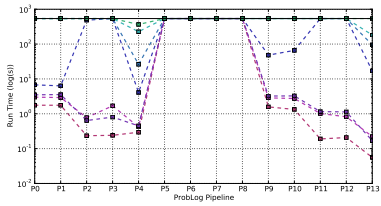


Figure B.5: Run times for the Alzheimer set, query $q5$.

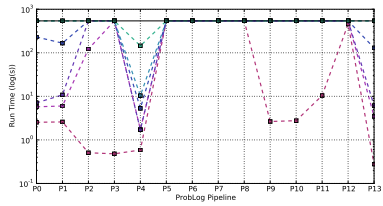


Figure B.6: Run times for the Alzheimer set, query $q6$.

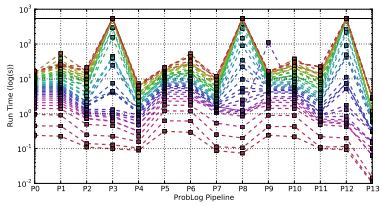


Figure B.7: Run times for the Balls set.

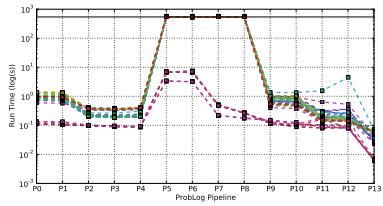


Figure B.8: Run times for the Dictionary set.

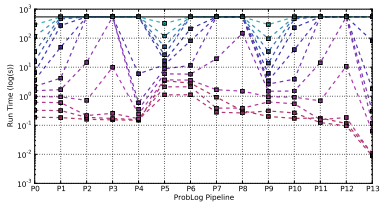


Figure B.9: Run times for the Grid set.

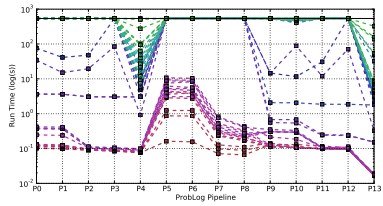


Figure B.10: Run times for the Les Miserables set.

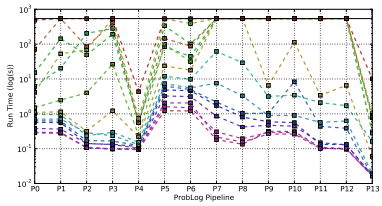


Figure B.11: Run times for the Smokers set.

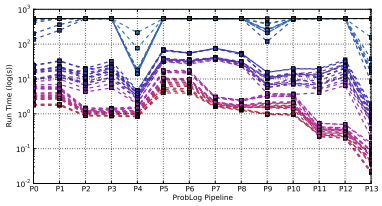


Figure B.12: Run times for the WebKB set.

COND Inference

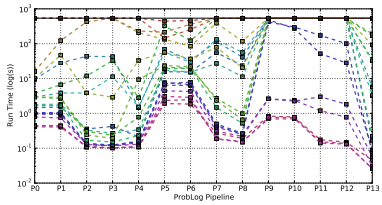


Figure B.13: Run times for the Smokers set.

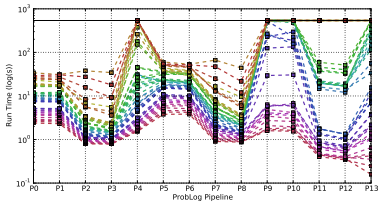


Figure B.14: Run times for the WebKB set.

B.0.2 Best-performing Pipelines

Tables B.1, B.2, B.3, B.4, B.5, B.6 and B.8 show an ascending ordering of the pipelines running MARG inference on the corresponding examples according to the (total) runtime for the "Alzheimer", "Balls", "Dictionary", "Grid", "Les Miserables", "Smokers" and "WebKB" benchmark sets. That is, 1^{st} indicates the pipeline that performs best (in lowest time); 2^{nd} indicates the second best pipeline, and so forth. Tables B.7 and B.9 show an ascending ordering of the pipelines running COND inference for the example programs in the "Smokers" and the "WebKB" benchmarks. For readability the index label $'P'$ is omitted, that is, each pipeline is associated only with a number. The empty cells indicate that no pipeline has successfully executed the inference task within the time out limit (540 seconds).

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
alzheimer_q1	graph05	4	13	3	0	2	1								
alzheimer_q1	graph06	4	13												
alzheimer_q1	graph07	4	13												
alzheimer_q1	graph08	4													
alzheimer_q2	graph05	13	2	3	4	1	0	10	9	11					
alzheimer_q2	graph06	4	13	1	0	2	10	9							
alzheimer_q2	graph07	4	1	0	13										
alzheimer_q2	graph08	4	13	1											
alzheimer_q2	graph09	4													
alzheimer_q2	graph10	4													
alzheimer_q2	graph11	4													
alzheimer_q2	graph12	4													
alzheimer_q2	graph13	4													
alzheimer_q3	graph01	13	3	2	11	12	4	10	1	9	0				
alzheimer_q3	graph05	13	11	12	10	9	3	2	1	0	4				
alzheimer_q3	graph06	13	12	11	10	9	3	2	0	1	4				
alzheimer_q3	graph07	13	11	12	10	9	2	3	0	1	4				
alzheimer_q3	graph08	13	12	11	10	9	4	3	1	2	0				
alzheimer_q3	graph09	13	12	10	11	9									
alzheimer_q3	graph11	11	13	9	12	10									
alzheimer_q4	graph05	13	12	11	3	2	10	4	9	1	0	6	5		
alzheimer_q4	graph06	13	12	3	2	10	9	4	11	1	0	6			
alzheimer_q4	graph07	13	12	11	3	2	10	4	9	1	0				
alzheimer_q4	graph08	13	12	11	3	2	10	4	9	1	0				
alzheimer_q4	graph09	13	12	11	3	2	4	10	9	1	0				
alzheimer_q4	graph10	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph11	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph12	13	11	12	3	2	10	9	4	1	0				
alzheimer_q4	graph13	13	12	11	3	10	2	9	4	1	0				
alzheimer_q4	graph14	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph15	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph16	13	12	11	3	10	2	9	4	1	0				
alzheimer_q4	graph17	13	12	11	10	3	2	9	4	1	0				
alzheimer_q4	graph18	13	11	12	10	3	2	9	4	1	0				
alzheimer_q4	graph19	13	12	11	10	3	2	9	4	1	0				
alzheimer_q4	graph20	13	11	12	10	3	2	9	4	1	0				
alzheimer_q5	graph05	13	11	12	2	3	4	10	9	1	0				
alzheimer_q5	graph06	13	4	2	12	11	3	10	9	1	0				
alzheimer_q5	graph07	13	4	2	3	12	11	9	10	0	1				
alzheimer_q5	graph08	4	1	0	13	9	10	2							
alzheimer_q5	graph09	4	13												
alzheimer_q5	graph10	13	4												
alzheimer_q5	graph12	4													
alzheimer_q6	graph05	13	3	2	4	0	1	9	10	11	12				
alzheimer_q6	graph06	4	13	0	1	2									
alzheimer_q6	graph07	4	13	0	1										
alzheimer_q6	graph08	4	13	1	0										
alzheimer_q6	graph09	4													
alzheimer_q6	graph10	4													

Table B.1: Ascending order of pipelines according to their total runtime for each program of the “Alzheimer” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
balls	test1	13	8	7	4	12	3	11	2	10	1	0	9	6	5
balls	test2	13	4	8	11	7	12	3	2	9	10	1	0	6	5
balls	test3	13	4	12	8	11	3	2	7	10	9	1	0	6	5
balls	test4	13	4	8	12	3	11	2	7	1	10	9	0	6	5
balls	test5	13	4	12	11	3	2	8	7	10	9	1	0	6	5
balls	test6	13	4	12	11	3	2	8	7	1	0	10	9	6	5
balls	test7	13	4	12	11	3	8	2	7	10	9	1	0	6	5
balls	test8	13	4	12	11	3	2	8	7	10	9	1	0	6	5
balls	test9	13	4	11	12	3	2	8	7	10	9	1	0	6	5
balls	test10	13	4	11	3	2	8	12	7	10	1	0	6	5	9
balls	test11	13	4	11	3	2	7	12	8	10	9	1	0	5	6
balls	test12	13	4	11	3	2	7	8	10	9	1	0	12	6	5
balls	test13	13	4	11	2	7	3	8	10	9	1	0	6	5	12
balls	test14	13	4	11	2	7	3	9	10	0	1	8	6	5	12
balls	test15	13	4	11	2	7	9	10	1	0	5	6	3	8	12
balls	test16	13	4	11	2	7	3	9	10	0	1	6	5	8	12
balls	test17	13	4	2	7	11	9	10	0	1	5	6	3	8	12
balls	test18	13	4	2	7	11	9	0	10	1	5	6	3	8	12
balls	test19	13	4	11	7	2	9	0	10	1	5	6	3	8	12
balls	test20	13	4	11	7	2	9	0	1	10	5	6	3	12	8
balls	test21	13	4	7	11	2	9	0	10	1	5	6	3	8	12
balls	test22	13	4	7	11	2	9	0	10	1	5	6	3	12	
balls	test23	13	4	7	2	11	9	0	1	10	5	6	3		
balls	test24	13	4	7	11	2	9	0	1	5	10	6	3		
balls	test25	13	4	7	11	2	9	0	5	10	1	6			
balls	test26	13	4	7	11	2	9	0	10	5	1	6			
balls	test27	13	4	7	2	11	9	0	5	6	1	10			
balls	test28	13	4	7	11	2	9	0	5	10	1	6			
balls	test29	13	4	7	11	9	2	0	5	1	10	6			
balls	test30	13	4	7	2	11	9	0	5	1	10	6			
balls	test31	13	4	7	2	11	9	0	5	10	6	1			
balls	test32	13	4	7	2	11	9	0	5	6	1	10			
balls	test33	13	4	7	11	2	9	0	5	10	6	1			
balls	test34	13	4	7	11	9	0	2	5	10	1	6			
balls	test35	13	4	7	11	9	0	2	5	1	6	10			
balls	test36	13	4	7	2	9	11	0	5	1	10	6			
balls	test37	13	4	7	11	9	0	2	5	10	6	1			
balls	test38	13	4	7	9	0	11	2	5	1	10	6			
balls	test39	13	4	7	9	0	2	5	11	1	10	6			
balls	test40	13	4	7	9	0	5	2	11	1	10	6			

Table B.2: Ascending order of pipelines according to their total runtime for each program of the “Balls” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
dictionary	q1	13	12	11	4	10	3	2	1	0	9	8	7	5	6
dictionary	q2	13	12	4	3	10	2	1	0	9	11	8	7	5	6
dictionary	q3	13	4	3	2	12	11	1	10	0	9	8	7	5	6
dictionary	q4	13	4	3	2	1	10	0	9	12	11	8	7	5	6
dictionary	q5	13	11	12	4	3	10	2	1	0	9	8	7	6	5
dictionary	q6	13	12	4	10	3	2	1	11	0	9	8	7	6	5
dictionary	q7	13	12	4	11	3	2	10	1	0	9	8	7	6	5
dictionary	q8	13	12	11	4	3	2	10	1	0	9	8	7	6	5
dictionary	q9	13	3	2	4	11	12	1	0	10	9				
dictionary	q10	13	3	4	12	2	11	10	9	1	0				
dictionary	q11	13	12	11	3	2	4	10	9	1	0				
dictionary	q12	13	12	11	3	2	4	10	9	1	0				
dictionary	q13	13	12	11	3	2	4	10	9	0	1				
dictionary	q14	13	12	11	3	2	4	10	9	1	0				
dictionary	q15	13	11	12	3	2	4	10	9	1	0	6			
dictionary	q16	13	12	11	3	2	4	10	9	1	0				
dictionary	q17	13	12	11	3	2	4	10	9	1	0				
dictionary	q18	13	12	11	3	2	4	10	9	1	0				
dictionary	q19	13	12	11	3	2	4	10	9	0	1				
dictionary	q20	13	3	2	4	1	0	10	9	11	12				
dictionary	q21	13	12	11	3	2	4	10	9	0	1				
dictionary	q22	13	3	11	4	12	2	10	9	1	0				
dictionary	q23	13	12	11	3	2	4	10	9	1	0				
dictionary	q24	13	12	11	3	2	4	10	9	1	0				
dictionary	q25	13	11	12	3	2	4	9	10	1	0				
dictionary	q26	13	11	12	4	2	3	10	1	9	0				
dictionary	q27	13	4	3	2	11	12	1	0	10	9				
dictionary	q28	13	12	11	3	2	4	10	9	1	0				
dictionary	q29	13	12	11	3	2	4	9	10	0	1				
dictionary	q30	13	11	12	3	2	4	10	9	0	1				
dictionary	q31	13	12	11	3	2	4	9	10	1	0				
dictionary	q32	13	12	11	4	3	2	10	9	1	0				
dictionary	q33	13	11	4	3	2	12	10	9	1	0				
dictionary	q34	13	12	11	3	2	4	10	9	1	0				
dictionary	q35	13	12	11	3	4	2	10	9	1	0				
dictionary	q36	13	12	11	3	2	4	10	9	1	0				
dictionary	q37	13	12	11	3	2	4	10	9	0	1				
dictionary	q38	13	3	12	2	4	11	1	0	10	9				
dictionary	q39	13	12	11	3	2	4	10	9	1	0				
dictionary	q40	13	12	11	3	2	4	10	9	1	0				
dictionary	q41	13	11	12	3	2	4	10	9	1	0				
dictionary	q42	13	12	11	3	2	4	10	9	1	0				
dictionary	q43	13	12	11	3	2	4	9	10	1	0				
dictionary	q44	13	3	2	4	11	12	1	0	10	9				
dictionary	q45	13	12	11	3	2	4	10	9	1	0				
dictionary	q46	13	12	11	3	2	4	9	10	0	1				
dictionary	q47	13	12	11	3	2	4	10	9	1	0				
dictionary	q48	13	12	11	3	2	4	10	9	1	0				
dictionary	q49	13	12	11	2	3	4	10	9	1	0				
dictionary	q50	13	4	3	2	12	11	1	0	10	9				
dictionary	q51	13	12	11	3	2	4	9	10	1	0				
dictionary	q52	13	12	11	3	2	4	9	10	1	0				
dictionary	q53	13	3	4	2	11	12	10	9	1	0				
dictionary	q54	13	12	11	3	4	2	0	1	10	9				
dictionary	q55	13	3	2	4	12	11	1	0	10	9				
dictionary	q56	13	12	11	3	2	4	10	9	1	0				
dictionary	q57	13	12	11	3	2	4	10	9	1	0				
dictionary	q58	13	12	11	3	4	2	10	9	0	1				
dictionary	q59	13	11	12	3	4	2	10	9	1	0				
dictionary	q60	13	12	11	3	2	4	10	9	0	1				
dictionary	q61	13	3	2	4	11	12	1	0	10	9				
dictionary	q62	13	12	11	3	2	4	10	9	1	0				
dictionary	q63	13	12	11	3	4	2	10	9	0	1				
dictionary	q64	13	12	11	4	3	2	9	10	0	1				

Table B.3: Ascending order of pipelines according to their total runtime for each program of the "Dictionary" benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
grid	test_1	13	12	4	3	2	10	11	1	0	9	8	7	6	5
grid	test_2	13	12	11	4	3	2	8	10	7	9	1	0	5	6
grid	test_3	13	11	4	12	2	3	8	7	10	1	0	9	5	6
grid	test_4	13	4	11	2	10	1	0	9	8	7	6	5	3	12
grid	test_5	13	4	9	10	0	1	6	5	11	2	7	8		
grid	test_6	13	4	0	9	10	1	5	6						
grid	test_7	13	9	0	4	5	10	1	6						
grid	test_8	9	0	5	13	6	10	1							
grid	test_9	9	0	5	1	10									
grid	test_10	0	9	5											
grid	test_11	9	0	5											
grid	test_12	0	9	5											

Table B.4: Ascending order of pipelines according to their total run time for each program of the “Grid” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
les_miserables	test_1	13	8	7	4	3	2	12	1	11	0	10	9	6	5
les_miserables	test_2	13	4	3	8	2	7	1	0	12	11	9	10	6	5
les_miserables	test_3	13	4	3	2	12	11	8	10	1	7	0	9	6	5
les_miserables	test_4	13	4	3	2	12	11	10	1	9	0	8	7	6	5
les_miserables	test_5	13	4	3	12	2	11	10	9	1	0	8	7	5	6
les_miserables	test_6	13	4	3	2	12	11	10	1	0	9	8	7	6	5
les_miserables	test_7	13	4	3	12	2	11	10	9	1	0	8	7	5	6
les_miserables	test_8	13	4	3	12	2	11	10	9	1	0	8	7	5	6
les_miserables	test_9	13	4	3	2	12	11	10	1	9	0	8	7	6	5
les_miserables	test_11	13	4	3	12	2	11	10	9	1	0	8	7	6	5
les_miserables	test_12	13	4	12	3	11	2	8	10	7	9	1	0	6	5
les_miserables	test_13	13	4	3	11	2	12	8	10	9	7	0	1	5	6
les_miserables	test_14	13	4	12	3	11	2	8	10	9	1	7	0	6	5
les_miserables	test_15	13	4	12	3	2	11	8	10	9	1	0	7	6	5
les_miserables	test_23	13	4	12	3	11	2	10	9	8	1	0	7	6	5
les_miserables	test_24	13	12	11	10	9	3	4	2	1	0				
les_miserables	test_25	13	12	11	9	10	4	3	2	0	1				
les_miserables	test_26	13	4	11	9	1	2	0	12	3	10				
les_miserables	test_27	13	4												
les_miserables	test_28	13	4	10											
les_miserables	test_29	13	4	10	9										
les_miserables	test_30	13	10	9	4	11	1	2	0						
les_miserables	test_31	13	4	9											
les_miserables	test_32	13	12	11	10	9									
les_miserables	test_33	13	4	10											
les_miserables	test_34	13	4												
les_miserables	test_35	13	4	10											
les_miserables	test_36	13	4												
les_miserables	test_37	13	4	10											
les_miserables	test_38	4	13	10											
les_miserables	test_39	13	4	10											
les_miserables	test_40	13	4	10											
les_miserables	test_41	13	4	10											
les_miserables	test_42	13	4												
les_miserables	test_43	13	4	10											
les_miserables	test_44	13	4	10											
les_miserables	test_45	13	4	10											
les_miserables	test_48	13	4	10											
les_miserables	test_49	13	4	10											
les_miserables	test_50	13	4												
les_miserables	test_51	13	4	9											
les_miserables	test_52	4	13												
les_miserables	test_53	13	4												
les_miserables	test_54	13	4												
les_miserables	test_55	13	4												

Table B.5: Ascending order of pipelines according to their total runtime for each program of the “Les Miserables” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
smokers	smokers-3-6	13	4	3	12	2	11	8	7	1	0	10	9	6	5
smokers	smokers-4-8	13	4	12	3	11	2	8	10	1	0	9	7	6	5
smokers	smokers-5-10	13	4	12	3	2	11	1	0	8	10	9	7	6	5
smokers	smokers-6-12	13	4	3	11	12	2	8	7	10	9	1	0	5	6
smokers	smokers-7-14	13	4	3	2	12	11	1	0	8	9	7	6	5	10
smokers	smokers-8-16	13	4	11	12	3	2	1	10	0	9	8	7	6	5
smokers	smokers-9-18	13	4	3	2	11	12	0	1	9	10	8	6	5	7
smokers	smokers-10-20	4	13	2	3	1	0	12	11	9	10	6	5	8	7
smokers	smokers-11-22	4	13	0	2	6	1	3	5						
smokers	smokers-12-24	13	4	0	1	6	5	2	3						
smokers	smokers-13-26	4	13	0	1	2	6	5							
smokers	smokers-14-28	4	13	0	1	2	3	6	5						
smokers	smokers-15-30	4	13	0	2	6	5	3							
smokers	smokers-16-32	13	4	2	0	1	3	11	9	12	6	5	10		
smokers	smokers-17-34	4	13	0	6										

Table B.6: Ascending order of pipelines according to their total runtime for each program of the “Smokers” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
smokers	smokers-3-6	13	3	4	2	12	11	8	7	1	0	10	9	6	5
smokers	smokers-4-8	13	3	2	4	12	11	8	7	1	0	9	10	5	6
smokers	smokers-5-10	13	3	2	4	8	7	1	12	0	11	10	9	6	5
smokers	smokers-6-12	13	3	2	4	12	11	8	7	1	0	10	9	6	5
smokers	smokers-7-14	13	3	2	4	8	7	0	1	6	5	12	11	10	9
smokers	smokers-8-16	13	3	2	4	8	7	1	0	12	10	9	11	6	5
smokers	smokers-9-18	13	3	2	4	8	7	0	1	5	6	12	11	10	9
smokers	smokers-10-20	13	4	2	3	0	1	6	5	8	7				
smokers	smokers-11-22	4	0	2	1	3	6	13	8	5	7				
smokers	smokers-12-24	4	13	0	1	6	8	3	2	5	7				
smokers	smokers-13-26	4	0	1	2	3	6	13	5						
smokers	smokers-14-28	3	2	4	0	1	13	8	6	5	7				
smokers	smokers-15-30	3	2	4	8	1	7	0	6	5	13				
smokers	smokers-16-32	3	2	13	4	8	7	1	0	5	6				
smokers	smokers-17-34	3	2	8	7	1	0	4	6	5	13				
smokers	smokers-18-36	3	2	0	8	4	6	1	5	7	13				
smokers	smokers-19-38	6	8	5	4	7									
smokers	smokers-20-40	5	6												
smokers	smokers-21-42	5	6												
smokers	smokers-22-44	0	1	5	4	6	2								

Table B.7: Ascending order of pipelines according to their total runtime for each program of the “Smokers” benchmark set executing the COND task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
webkb	webkb-3	13	12	11	4	3	2	10	9	8	7	1	0	6	5
webkb	webkb-4	13	12	11	4	3	2	10	9	8	7	1	0	6	5
webkb	webkb-5	13	12	11	4	3	2	9	10	8	7	0	1	6	5
webkb	webkb-6	13	12	11	3	4	2	8	10	9	7	0	1	6	5
webkb	webkb-7	13	12	11	4	3	2	10	9	8	7	1	0	5	6
webkb	webkb-8	13	12	11	4	3	2	8	9	10	7	0	1	6	5
webkb	webkb-9	13	12	11	4	3	2	10	9	8	7	1	0	5	6
webkb	webkb-10	13	12	11	3	2	4	10	9	8	7	0	1	5	6
webkb	webkb-11	13	12	11	4	3	2	8	10	7	9	0	1	5	6
webkb	webkb-12	13	12	11	3	4	2	8	10	9	7	0	1	5	6
webkb	webkb-13	13	12	11	3	4	2	8	9	10	7	1	0	6	5
webkb	webkb-14	13	12	11	3	2	4	8	7	9	10	0	1	6	5
webkb	webkb-15	13	12	11	3	2	4	8	9	10	7	1	0	6	5
webkb	webkb-16	13	12	11	3	2	4	8	10	9	7	1	0	6	5
webkb	webkb-17	13	12	11	3	2	4	8	7	10	9	0	1	6	5
webkb	webkb-18	13	12	11	3	2	4	8	7	10	9	1	0	6	5
webkb	webkb-19	13	11	12	3	2	4	8	7	9	10	1	0	6	5
webkb	webkb-20	13	12	11	3	2	4	8	7	10	9	1	0	5	6
webkb	webkb-21	13	12	11	3	2	4	8	7	10	9	1	0	5	6
webkb	webkb-22	13	4	11	2	9	10	3	12	0	1	8	6	5	7
webkb	webkb-23	13	4	10	9	11	2	3	12	0	1	8	6	5	7
webkb	webkb-24	13	4	9	10	11	2	0	1	3	12	8	6	5	7
webkb	webkb-25	13	4	9	11	10	2	0	1	3	12	8	6	5	7
webkb	webkb-26	13	4	9	11	2	0	10	3	12	1	8	6	5	7
webkb	webkb-27	13	4	9	10	11	0	2	1	3	12	8	6	5	7
webkb	webkb-28	13	4	11	2	9	10	0	3	12	1	8	6	5	7
webkb	webkb-29	13	4	9	11	10	2	0	1	12	3	8	6	5	7
webkb	webkb-30	13	4	9	2	11	10	0	3	1	12	8	5	6	7
webkb	webkb-31	13	4	11	9	10	2	0	3	1	12	8	6	5	7
webkb	webkb-32	13	4	9	11	10	2	0	3	1	12	8	6	5	7
webkb	webkb-33	13	4	9	10	11	2	0	1	3	12	8	6	5	7
webkb	webkb-34	13	4	2	9	11	10	0	3	12	1	8	6	5	7
webkb	webkb-35	13	4	9	2	11	10	3	0	12	1	8	6	5	7
webkb	webkb-36	13	4	9	11	10	2	0	1	3	12	8	6	5	7
webkb	webkb-37	13	4	9	0										
webkb	webkb-38	13	4	9	0	1	10								
webkb	webkb-39	13	4	0	9	1									
webkb	webkb-40	13	4	9	0										
webkb	webkb-41	4	13	9	0										
webkb	webkb-42	4	13	9											
webkb	webkb-43	4	13	9											
webkb	webkb-44	4	13	9											
webkb	webkb-46	4													

Table B.8: Ascending order of pipelines according to their total runtime for each program of the "WebKB" benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
webkb	webkb-3	13	12	11	3	2	8	7	4	10	9	1	0	5	6
webkb	webkb-4	13	12	11	3	2	8	7	10	4	9	1	0	6	5
webkb	webkb-5	12	13	11	3	2	8	7	4	9	10	0	1	6	5
webkb	webkb-6	12	11	13	3	2	8	7	4	10	9	1	0	6	5
webkb	webkb-7	13	12	11	3	2	8	7	4	10	9	1	0	6	5
webkb	webkb-8	12	11	13	3	2	8	7	4	10	9	1	0	6	5
webkb	webkb-9	12	11	3	2	8	7	13	4	1	0	10	9	6	5
webkb	webkb-10	12	11	3	2	13	8	7	4	10	9	1	0	5	6
webkb	webkb-11	12	11	3	2	13	8	7	4	10	9	0	1	6	5
webkb	webkb-12	12	11	3	2	8	7	13	4	1	0	10	9	6	5
webkb	webkb-13	12	11	3	2	8	7	13	4	0	1	9	10	6	5
webkb	webkb-14	12	11	3	2	8	7	4	13	1	0	9	10	6	5
webkb	webkb-15	12	11	3	2	8	7	13	4	1	0	6	5	9	10
webkb	webkb-16	12	11	3	2	8	7	4	13	0	1	6	5	9	10
webkb	webkb-17	12	3	2	11	8	7	4	1	0	13	6	5	10	9
webkb	webkb-18	12	3	2	11	8	7	4	13	1	0	6	5	9	10
webkb	webkb-19	3	2	12	8	11	7	4	13	1	0	6	5	10	9
webkb	webkb-20	3	2	12	8	11	7	4	1	0	13	6	5	10	9
webkb	webkb-21	3	12	2	8	11	7	4	1	0	13	6	5		
webkb	webkb-22	3	2	8	7	1	0	4	12	11	6	5	13		
webkb	webkb-23	3	2	8	7	4	1	0	12	11	6	5	13		
webkb	webkb-24	3	2	8	7	4	0	1	12	6	5	11	13	9	
webkb	webkb-25	3	2	8	7	1	0	12	11	6	5	4	13		
webkb	webkb-26	3	2	8	7	0	1	12	6	11	5	4	13		
webkb	webkb-27	3	2	8	7	1	0	12	4	11	6	5	13		
webkb	webkb-28	3	2	8	7	1	0	12	6	11	5	4	13		
webkb	webkb-29	3	2	8	7	1	0	4	12	11	6	5	13	10	
webkb	webkb-30	3	2	8	7	1	4	0	12	11	6	5	13		
webkb	webkb-31	3	2	8	7	1	0	12	4	11	6	5	13		
webkb	webkb-32	3	2	8	7	0	1	4	12	11	6	5	13		
webkb	webkb-33	3	2	8	7	1	0	6	12	5	11	4			
webkb	webkb-34	3	2	8	7	1	0	4	5	6	12	11	13		
webkb	webkb-35	3	2	8	7	1	0	4	6	5	12	11			
webkb	webkb-36	3	2	8	7	1	0	6	5	4	12	11			
webkb	webkb-37	3	2	8	7	0	1	6	5	4					
webkb	webkb-38	3	2	8	7	1	0	6	5	4					
webkb	webkb-39	3	2	8	7	1	0	6	5						
webkb	webkb-40	3	2	8	7	0	1	6	5	4					
webkb	webkb-41	3	2	8	7	1	0	6	5						
webkb	webkb-42	3	2	8	7	1	0	6	5	4					
webkb	webkb-43	3	2	8	7	1	0	6	5						
webkb	webkb-44	3	2	8	7	1	0	6	5	4					
webkb	webkb-45	1	0	3	2	8	6	5	7						
webkb	webkb-46	3	2	8	7	1	0	6	5						
webkb	webkb-47	3	8	2	7	1	0	6	5						
webkb	webkb-48	8	2	3	7	1	0	6	5						
webkb	webkb-49	3	8	2	7	1	0	6	5						
webkb	webkb-50	3	8	2	0	1	7	6	5						

Table B.9: Ascending order of pipelines according to their total runtime for each program of the "WebKB" benchmark set executing the COND task.

Table B.10 and Table B.12 summarize the order results from the previous tables and show for the total number of benchmark programs for which each pipeline performed best, second best and so forth for the MARG task and for the COND task respectively. Table B.11 and Table B.13 show the number of benchmark programs for which each pipeline performs best, second best and so forth relative

to the total number of programs in a benchmark set for the MARG task and the COND task respectively. For example, pipeline $P8$ performs second best for two benchmarks – one from the "Balls" and one from the "Les Miserables" sets. There are 40 and 45 benchmarks which have been successfully executed by at least one pipeline in the "Balls" and "Les Miserables" sets respectively. Then we compute the relative number of programs for which $P8$ performs second best as $1/40 + 1/45 = 0.05$. The relative number shown in Table B.11 for pipeline $P4$ performing first is 3.71. This states that in 3.71 out of 12 benchmark sets this pipeline performs best, that is in 31% of the cases $P4$ performs best. The sum of the ratios in the last rows of Table B.11 and Table B.13 is smaller or equal to the number of benchmark sets used for experimenting. For Table B.11 this number is smaller or equal than 12 because for the "Alzheimer" benchmark set we used 6 instances.

The total and the relative (to the total number of programs in a benchmark set) number of timeouts for a pipeline executing the MARG and the COND tasks are shown in Table B.14 and Table B.15 respectively.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	2	0	0	0	33	0	0	0	0	3	0	1	0	229
2^{th}	3	2	1	11	115	0	0	0	2	3	1	16	82	22
3^{th}	13	3	14	21	8	5	0	22	2	22	16	93	26	0
4^{th}	8	6	32	77	26	0	1	4	2	11	15	26	12	3
5^{th}	5	8	85	28	9	1	4	5	0	15	15	27	13	0
6^{th}	8	7	52	14	50	3	1	3	2	18	21	21	9	0
7^{th}	32	16	16	9	5	3	2	0	29	25	63	3	4	0
8^{th}	16	25	4	7	11	16	2	17	1	62	33	5	3	0
9^{th}	28	80	1	6	0	2	2	3	10	18	33	5	8	0
10^{th}	67	35	1	1	3	8	5	15	0	32	15	3	10	0
11^{th}	17	24	0	0	0	1	25	3	34	4	6	0	0	0
12^{th}	23	9	0	9	0	5	19	21	1	6	2	0	1	0
13^{th}	0	0	0	1	0	38	43	0	7	0	0	0	2	0
14^{th}	0	0	0	0	0	42	19	17	1	1	1	0	9	0
Sum:	222	215	206	184	260	124	123	110	91	220	221	200	179	254
Maximum possible value:				268										
Percentage:	83%	80%	77%	69%	97%	46%	46%	41%	34%	82%	82%	75%	67%	95%

Table B.10: The number of benchmark programs for which a pipeline performs best, second best and so forth without timing out and executing the MARG task. The total number of programs shows how many programs have been executed with the corresponding pipeline and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	0.17	0	0	0	3.71	0	0	0	0	0.25	0	0.14	0	7.73
2^{th}	0.25	0.25	0.11	0.45	3.6	0	0	0	0.05	0.25	0.02	0.85	2.56	2.3
3^{th}	1.18	0.39	0.88	0.98	0.26	0.42	0	0.55	0.05	0.69	0.48	2.74	1.11	0
4^{th}	0.69	0.62	1.1	2.21	0.86	0	0.07	0.1	0.05	0.32	1.03	0.91	0.66	0.34
5^{th}	0.32	0.34	2.81	0.99	0.16	0.08	0.28	0.12	0	1.07	0.75	0.81	0.6	0
6^{th}	0.31	0.76	1.67	0.82	1.29	0.16	0.07	0.07	0.05	0.49	1.17	0.7	0.21	0
7^{th}	0.86	0.55	0.77	0.54	0.27	0.17	0.15	0	0.93	1.31	1.63	0.17	0.14	0
8^{th}	0.64	1.0	0.09	0.21	0.69	0.54	0.17	0.56	0.03	1.72	1.12	0.19	0.07	0
9^{th}	0.87	2.8	0.14	0.14	0	0.05	0.05	0.13	0.38	0.61	0.84	0.4	0.22	0
10^{th}	2.47	0.93	0.08	0.02	0.43	0.2	0.17	0.41	0	0.9	0.49	0.05	0.37	0
11^{th}	0.46	0.68	0	0	0	0.07	0.79	0.17	0.87	0.18	0.19	0	0	0
12^{th}	0.65	0.21	0	0.22	0	0.26	0.53	0.61	0.08	0.24	0.09	0	0.03	0
13^{th}	0	0	0	0.08	0	1.11	1.21	0	0.22	0	0	0	0.05	0
14^{th}	0	0	0	0	0	1.19	0.57	0.48	0.03	0.03	0.07	0	0.28	0
Sum	8.87	8.53	7.65	6.66	11.27	4.25	4.06	3.2	2.74	8.06	7.88	6.96	6.3	10.37
Maximum possible value:				12.00										
Percentage	74%	71%	64%	56%	94%	35%	34%	27%	23%	67%	66%	58%	53%	86%

Table B.11: The relative number of benchmark programs to the total number of benchmarks in a set for which a pipeline performs best, second best and so forth without timing out and executing the MARG task. The total number of programs shows how many programs have been executed with the corresponding pipeline relative to the total number of programs in the benchmark set, and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	1	1	0	35	3	2	1	0	1	0	0	0	13	11
2^{th}	3	1	32	9	1	0	2	0	4	0	0	10	4	2
3^{th}	2	1	14	10	3	2	0	0	25	0	0	4	2	3
4^{th}	2	2	11	7	9	0	0	28	5	0	0	2	0	0
5^{th}	5	25	6	2	3	0	2	1	14	0	0	3	3	2
6^{th}	22	6	1	0	1	0	4	18	9	0	0	3	0	1
7^{th}	5	5	0	1	13	1	16	8	4	0	0	0	5	6
8^{th}	2	5	1	0	13	18	6	4	1	0	1	1	8	4
9^{th}	6	9	0	0	7	9	4	1	1	1	6	9	1	0
10^{th}	9	2	0	0	0	5	9	4	0	7	2	2	3	6
11^{th}	2	6	0	0	4	7	7	0	0	4	5	4	2	0
12^{th}	6	2	0	0	0	7	0	0	0	5	3	3	0	12
13^{th}	0	0	0	0	0	3	14	0	0	4	6	0	0	0
14^{th}	0	0	0	0	0	14	3	0	0	5	3	0	0	0
Sum:	65	65	65	64	57	68	68	64	64	26	26	41	41	47
Maximum possible value:				68										
Percentage:	96%	96%	96%	94%	84%	100%	100%	94%	94%	38%	38%	60%	60%	69%

Table B.12: The number of benchmark programs for which a pipeline performs best, second best and so forth without timing out and executing the COND task. The total number of programs shows how many programs have been executed with the corresponding pipeline and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	0.05	0.02	0	0.88	0.15	0.1	0.05	0	0.02	0	0	0	0.27	0.46
2^{th}	0.12	0.05	0.81	0.39	0.05	0	0.1	0	0.11	0	0	0.21	0.08	0.07
3^{th}	0.1	0.05	0.52	0.21	0.15	0.1	0	0	0.55	0	0	0.08	0.04	0.09
4^{th}	0.07	0.1	0.29	0.17	0.45	0	0	0.61	0.16	0	0	0.04	0	0
5^{th}	0.13	0.61	0.12	0.1	0.09	0	0.1	0.05	0.44	0	0	0.06	0.15	0.04
6^{th}	0.49	0.15	0.05	0	0.02	0	0.17	0.55	0.22	0	0	0.15	0	0.05
7^{th}	0.19	0.22	0	0.05	0.3	0.02	0.36	0.17	0.2	0	0	0	0.1	0.18
8^{th}	0.1	0.16	0.05	0	0.27	0.46	0.21	0.17	0.05	0	0.02	0.02	0.2	0.08
9^{th}	0.15	0.27	0	0	0.15	0.39	0.11	0.05	0.05	0.02	0.12	0.19	0.05	0
10^{th}	0.27	0.04	0	0	0	0.13	0.25	0.2	0	0.15	0.07	0.07	0.06	0.21
11^{th}	0.04	0.12	0	0	0.08	0.15	0.15	0	0	0.14	0.19	0.08	0.1	0
12^{th}	0.12	0.04	0	0	0	0.15	0	0	0	0.19	0.09	0.15	0	0.25
13^{th}	0	0	0	0	0	0.09	0.41	0	0	0.08	0.18	0	0	0
14^{th}	0	0	0	0	0	0.41	0.09	0	0	0.16	0.06	0	0	0
Sum:	1.83	1.83	1.84	1.8	1.71	2	2	1.8	1.8	0.74	0.73	1.05	1.05	1.43
Maximum possible value:				2.00										
Percentage:	92%	92%	92%	90%	86%	100%	100%	90%	90%	37%	37%	53%	53%	72%

Table B.13: The relative number of benchmark programs to the total number of benchmarks in a set for which a pipeline performs best, second best and so forth without timing out and executing the COND task. The total number of programs shows how many programs have been executed with the corresponding pipeline relative to the total number of programs in the benchmark set, and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	46	53	62	84	8	144	145	158	177	48	47	68	89	14
Total (relative):	3.14	3.48	4.35	5.34	0.72	7.76	7.94	8.8	9.28	3.95	4.12	5.04	5.71	1.64

Table B.14: Total and relative (to the total number of benchmarks in a set) number of benchmark programs for which MARG inference times out.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	3	3	3	4	11	0	0	4	4	42	42	27	27	21
Total (relative):	0.15	0.15	0.15	0.2	0.29	0.0	0.0	0.2	0.2	1.25	1.25	0.94	0.94	0.55

Table B.15: Total and relative (to the total number of benchmarks in a set) number of benchmark programs for which COND inference times out.

Appendix C

Compaction Statistics

Here we present the diagrams that illustrate the number of patterns that are detected and compacted per AND-OR graph size four the 7 benchmark sets. The benchmark sets are presented in Chapter 2.

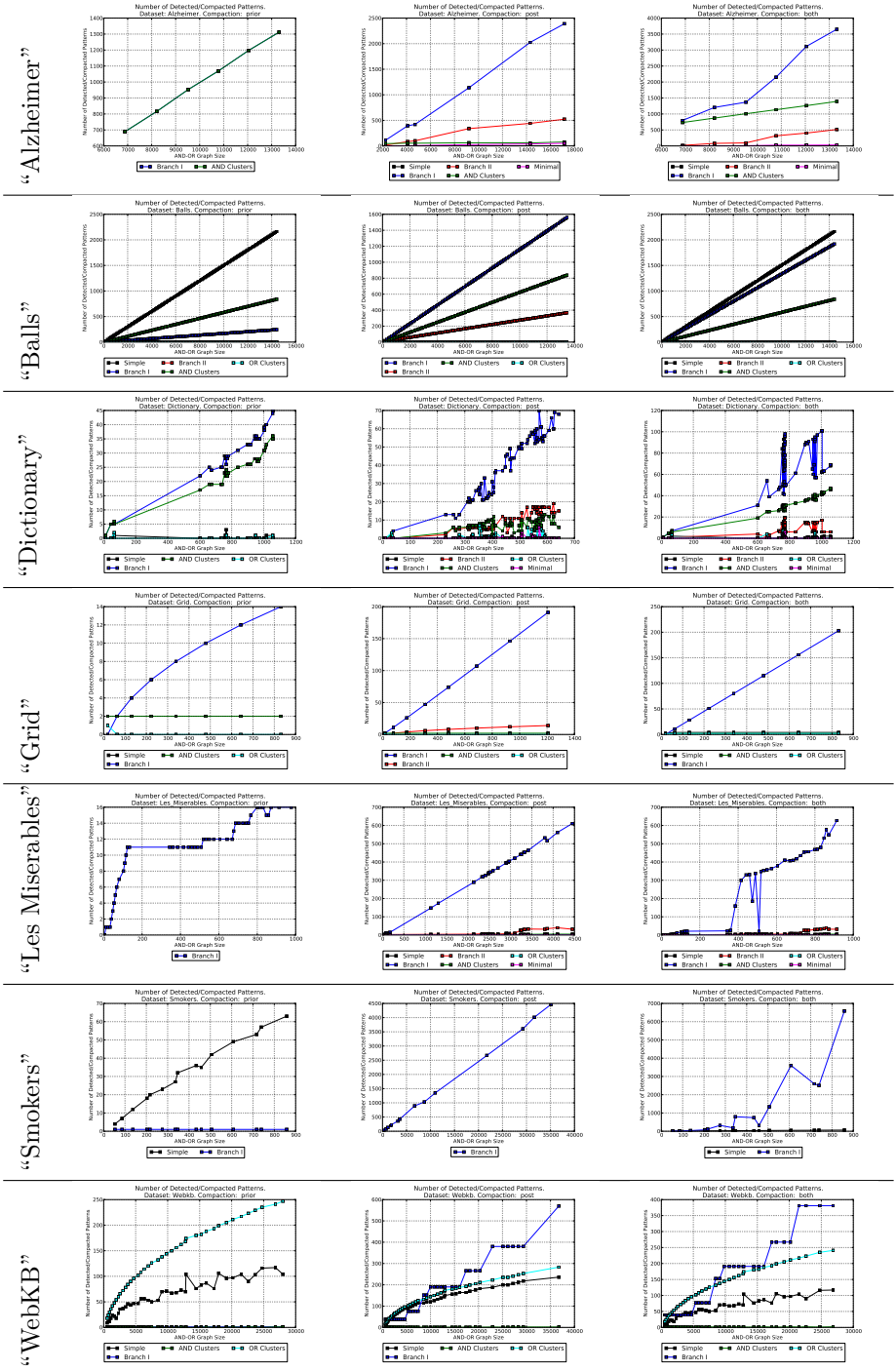


Figure C.1: Number of detected and compacted patterns per AND-OR graph size. Boolean formula preprocessing method: recursive node merging.

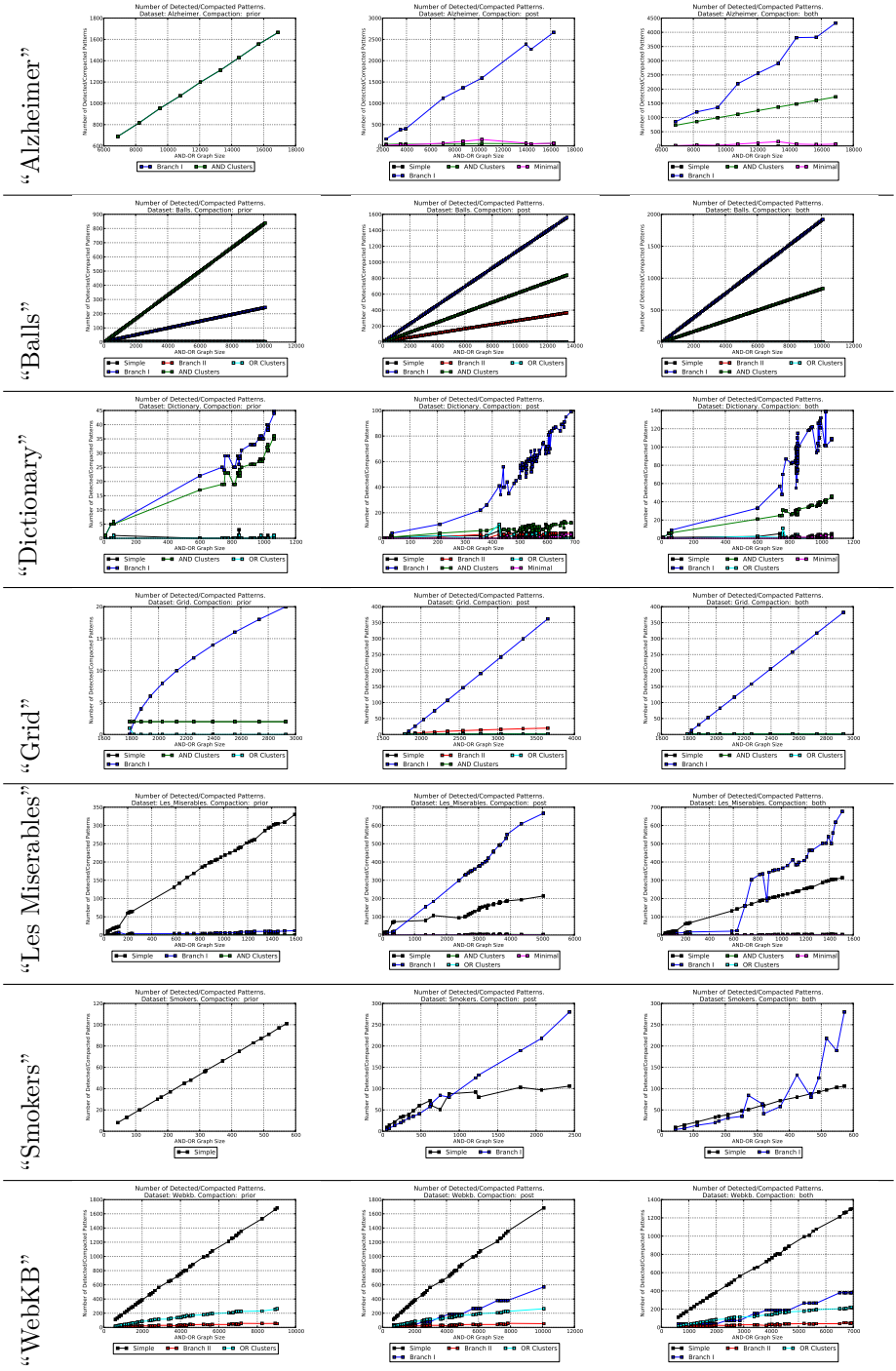


Figure C.2: Number of detected and compacted patterns per AND-OR graph size. Boolean formula preprocessing method: subformulae repetition detection.

Bibliography

[1] Aloul, F. A., Markov, I. L., and Sakallah, K. A. Faster SAT and smaller BDDs via common function structure. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design, ICCAD '01* (2001), IEEE Press, pp. 443–448.

[2] Bacchus, F., and Winter, J. Effective preprocessing with hyper-resolution and equality reduction. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Selected Revised Papers* (May 5-8 2003), pp. 341–355.

[3] Bragaglia, S., and Riguzzi, F. Approximate inference for logic programs with annotated disjunctions. In *Inductive Logic Programming - 20th International Conference, ILP 2010, Revised Papers* (June 27-30 2010), pp. 30–37.

[4] Bruynooghe, M., Mantadelis, T., Kimmig, A., Gutmann, B., Vennekens, J., Janssens, G., and De Raedt, L. ProbLog technology for inference in a probabilistic First Order logic. In *Proceedings of the 19th European Conference on Artificial Intelligence, ECAI 2010* (August 16-20 2010), H. Coelho, R. Studer, and M. Woolridge, Eds., IOS Press, pp. 719–724.

[5] Bryant, R. E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35* (1986), 677–691.

[6] Chen, W., Swift, T., and Warren, D. S. Efficient top-down computation of queries under the well-founded semantics. *The Journal of Logic Programming 24*, 3 (September 1995), 161–199.

[7] Codognet, P., and Diaz, D. *The Journal of Logic Programming 27*, 3 (1996), 185–226.

[8] Darwiche, A. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, AAAI/IAAI 2002* (July 28 - August 1 2002), R. Dechter and R. S. Sutton, Eds., AAAI Press/MIT Press, pp. 627–634.

[9] Darwiche, A. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004* (August 22-27 2004), pp. 328–332.

[10] Darwiche, A. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[11] Darwiche, A. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011* (July 16-22 2011), pp. 819–826.

[12] Darwiche, A., and Marquis, P. A knowledge compilation map. *Journal of Artificial Intelligence Research 17* (2002), 229–264.

[13] De Raedt, L., Kimmig, A., and Toivonen, H. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (2007), AAAI Press, pp. 2468–2473.

[14] Fierens, D., den Broeck, G. V., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G., and Raedt, L. D. Inference and learning in probabilistic logic programs using weighted boolean formulas. *TPLP 15*, 3 (2015), 358–401.

[15] Fierens, D., Van den Broeck, G., Bruynooghe, M., and De Raedt, L. Constraints for probabilistic logic programming. In *Proceedings of the NIPS Probabilistic Programming Workshop* (December 7-8 2012), D. Roy, V. Mansinghka, and N. Goodman, Eds., p. 4 pages.

[16] Fierens, D., Van den Broeck, G., Thon, Ingo, Gutmann, Bernd, and De Raedt, L. Inference in probabilistic logic programs using weighted CNFs. In *Proceedings of the 27th Annual Conference on Uncertainty in Artificial Intelligence, UAI11* (July 14-17 2011), F. Gagliardi Cozman and A. Pfeffer, Eds., AUAI Press, pp. 211–220.

[17] Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI 99* (July 31 - August 6 1999), pp. 1300–1309.

[18] Frühwirth, T. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming 37*, 1-3 (October 1998), 95–138.

[19] Frühwirth, T. *Constraint Handling Rules*. Cambridge University Press, August 2009.

[20] Getoor, L., and Taskar, B. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[21] Gutmann, B. *On Continuous Distributions and Parameter Estimation in Probabilistic Logic Programs*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, October 2011. De Raedt, Luc (supervisor).

[22] Gutmann, B., Jaeger, M., and De Raedt, L. Extending ProbLog with continuous distributions. In *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP–10)* (2010), P. Frasconi and F. A. Lisi, Eds.

[23] Gutmann, B., Thon, I., and De Raedt, L. Learning the parameters of probabilistic logic programs from interpretations. In *Machine Learning and Knowledge Discovery in Databases, European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Athens, Greece, 5-9 September 2011* (2011), D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds., Springer, pp. 581–596.

[24] Hinman, P. G. *Fundamental of Mathematical Logic*. Peters, Wellesley, MA, 2005.

[25] Hintsanen, P. The most reliable subgraph problem. 2007, pp. 471–478.

[26] Huang, J. Solving MAP exactly by searching on compiled arithmetic circuits. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)* (2006), pp. 143–148.

[27] Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C. The CLP(R) language and system. *ACM Trans. Program. Lang. Syst. 14*, 3 (may 1992), 339–395.

[28] Janhunen, T. Representing normal programs with clauses. In *Proceedings of the 16th European Conference on Artificial Intelligence* (2004), IOS Press, pp. 358–362.

[29] Kersting, K., and De Raedt, L. Bayesian logic programming: Theory and tool. In *Introduction to Statistical Relational Learning*. MIT Press, 2007, p. 291.

[30] Kimmig, A. *A Probabilistic Prolog and its Applications*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, November 2010. De Raedt, Luc (supervisor).

[31] Kimmig, A., and Costa, F. Link and node prediction in metabolic networks with probabilistic logic. In *Bisociative Knowledge Discovery*, M. Berthold and M. R. Berthold, Eds., vol. 7250 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 407–426.

[32] Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., and Rocha, R. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming 11* (2011), 235–262.

[33] Kimmig, A., Gutmann, B., and Santos Costa, V. Trading memory for answers: Towards tabling ProbLog. In *International Workshop on Statistical Relational Learning, Leuven, Belgium, 2-4 July 2009* (July 2009), P. Domingos and K. Kersting, Eds.

[34] Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., and De Raedt, L. On the efficient execution of ProbLog programs. In *Logic Programming* (2008), M. Garcia de la Banda and E. Pontelli, Eds., vol. 5366 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 175–189.

[35] Kimmig, A., Van den Broeck, G., and De Raedt, L. An algebraic Prolog for reasoning about possible worlds. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence* (7-11 August 2011), W. Burgard and D. Roth, Eds., AAAI Press, pp. 209–214.

[36] Knuth, D. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, MA, 1993.

[37] Koller, D., and Friedman, N. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[38] Kowalski, R. A. Predicate logic as programming language. In *IFIP Congress* (1974), pp. 569–574.

[39] Lagniez, J., and Marquis, P. Preprocessing for propositional model counting. In *28th AAAI Conference on Artificial Intelligence* (27–31 July 2014), pp. 2688–2694.

[40] Lloyd, J. W. *Foundations of Logic Programming; (2Nd Extended Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[41] Mantadelis, T. *Efficient Algorithms for Prolog Based Probabilistic Logic Programming*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, November 2012. Janssens, Gerda (supervisor).

[42] Mantadelis, T., Demoen, B., and Janssens, G. A simplified fast interface for the use of CUDD for binary decision diagrams, 2008. `http://people.cs.kuleuven.be/~theofrastos.mantadelis/tools/simplecudd.html`.

[43] Mantadelis, T., and Janssens, G. Dedicated tabling for a probabilistic setting. In *Technical Communications of the 26th International Conference on Logic Programming, International Conference on Logic Programming* (16-19 July 2010), M. V. Hermenegildo and T. Schaub, Eds., vol. 7 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 124–133.

[44] Mantadelis, T., and Janssens, G. Variable compression in ProbLog. In *Lecture Notes in Computer Science, Logic for Programming, Artificial Intelligence and Reasoning (LPAR)* (10-15 October 2010), C. Fermüller and A. Voronkov, Eds., Springer-Verlag's, pp. 504–518.

[45] Mantadelis, T., and Janssens, G. Nesting probabilistic inference. In *CICLOPS, International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), Lexington, Kentucky, USA, 10 July 2011* (15-16 July 2011), S. Abreu and V. Santos Costa, Eds., pp. 1–16.

[46] Mantadelis, T., Rocha, R., Kimmig, A., and Janssens, G. Preprocessing Boolean formulae for BDDs in a probabilistic context. In *Proceedings if the 12th European Conference on Logics in Artificial Intelligence, Logics in Artificial Intelligence, JELIA 2010* (13-15 September 2010), T. Janhunen and I. Niemelä, Eds., Springer, pp. 260–272.

[47] Mantadelis, T., Shterionov, D., and Janssens, G. Compacting boolean formulae for inference in probabilistic logic programming. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)* (27-30 September 2015), p. 13. To appear.

[48] Mariën, M., Wittocx, J., and Denecker, M. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT* (2006), pp. 19–34.

[49] Meert, W. *Inference and Learning for Directed Probabilistic Logic Models.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, 2011. Blockeel, Hendrik (supervisor).

[50] Meert, W., Struyf, J., and Blockeel, H. CP-logic theory inference with contextual variable elimination and comparison to BDD based inference methods. 96–109.

[51] Michels, S., Hommersom, A., Lucas, P. J. F., Velikova, M., and Koopman, P. W. M. Inference for a new probabilistic constraint logic. In *IJCAI* (2013), F. Rossi, Ed., IJCAI/AAAI.

[52] Moldovan, B., M. van Otterlo, P. Moreno, J. S.-V., and Raedt, L. D. Statistical relational learning of object affordances for robotic manipulation. In *Latest Advances in Inductive Logic Programming, International Conference on Inductive Logic Programming* (31 July - 3 August 2011 2012), Imperial College Press, p. 6.

[53] Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., and Troyansky, L. Determining computational complexity from characteristic 'phase transitions'. *Nature 400*, 6740 (1999), 133–137.

[54] Muise, C., Mcilraith, S. A., Beck, J. C., and Hsu, E. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence* (2012).

[55] Narodytska, N., and Walsh, T. Constraint and variable ordering heuristics for compiling configuration problems. In *Proceedings of the 20th international joint conference on Artifical intelligence, IJCAI'07* (2007), Morgan Kaufmann Publishers Inc., pp. 149–154.

[56] Panda, S., and Somenzi, F. Who are the variables in your neighborhood. In *ICCAD* (1995), pp. 74–77.

[57] Paridel, K., Mantadelis, T., Yasar, A.-U.-H., Preuveneers, D., Janssens, G., Vanrompay, Y., and Berbers, Y. Analyzing the efficiency of context-based grouping on collaboration in VANETs with large-scale simulation. *Journal of Ambient Intelligence and Humanized Computing 5*, 4 (Aug. 2014), 475–490.

[58] Poole, D. Logic programming, abduction and probability. *New Generation Computing 11* (1993), 377–400.

[59] Raedt, L. D., Frasconi, P., Kersting, K., and Muggleton, S., Eds. *Probabilistic Inductive Logic Programming - Theory and Applications* (March 3 2008), vol. 4911 of *Lecture Notes in Computer Science*, Springer.

[60] Rauzy, A., Châtelet, E., Dutuit, Y., and Bérenguer, C. A practical comparison of methods to assess sum-of-products. *Rel. Eng. & Sys. Safety 79*, 1 (2003), 33–42.

[61] Richardson, M., and Domingos, P. Markov logic networks. *Machine Learning 62*, 1-2 (2006), 107–136.

[62] Rocha, R., Silva, F., and Santos Costa, V. YapTab: A tabling engine designed to support parallelism. In *Proceedings of the 2nd Workshop on Tabulation in Parsing and Deduction, TAPD'2000* (Vigo, Spain, September 2000), pp. 77–87.

[63] Rudell, R. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design* (Los Alamitos, CA, USA, 1993), ICCAD '93, IEEE Computer Society Press, pp. 42–47.

[64] Santos Costa, V., and Cussens, J. CLP(BN): Constraint logic programming for probabilistic knowledge. In *In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI03)* (2003), Morgan Kaufmann, pp. 517–524.

[65] Sato, T. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)* (1995), MIT Press, pp. 715–729.

[66] Sato, T., and Kameya, Y. PRISM: A language for symbolic-statistical modeling. In *In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97* (1997), pp. 1330–1335.

[67] Sato, T., and Kameya, Y. A Viterbi-like algorithm and EM learning for statistical abduction. In *Proceedings of Workshop on Fusion of Domain Knowledge with Data for Decision Support* (2000).

[68] Sato, T., and Kameya, Y. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research 15* (2001), 391–454.

[69] Shterionov, D., and Janssens, G. Data acquisition and modeling for learning and reasoning in probabilistic logic environment. In *Proceedings of the 15th Portuguese Conference on Artificial Intelligence, Portuguese Conference on Artificial Intelligence (EPIA 2011)* (10-13 October 2011), L. Antunes, H. S. Pinto, R. Prada, and P. Trigo, Eds., pp. 298–312.

[70] Shterionov, D., and Janssens, G. cProbLog: Restricting the possible worlds of probabilistic logic programs. In *Proceedings FLOC Workshop Probabilistic Logic Programming (PLP 2014)* (17 July 2014), p. 12.

[71] Shterionov, D., and Janssens, G. Crucial components in probabilistic inference pipelines. In *Proceedings of the 30th Symposium On Applied Computing (SAC 2015)* (13-17 April 2015), ACM, pp. 1887–1889.

[72] Shterionov, D., and Janssens, G. Implementation and performance of probabilistic inference pipelines. In *Proceedings of the 17th International Symposium Practical Aspects of Declarative Languages, PADL 2015* (18-19 June 2015), vol. 9131 of *Lecture Notes in Computer Science*, pp. 90–104.

[73] Shterionov, D., Kimmig, A., Mantadelis, T., and Janssens, G. DNF sampling for ProbLog inference. In *Proceedings International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), Edinburgh, Scotland, 15 July 2010* (15 July 2010), p. 15.

[74] Shterionov, D., Mantadelis, T., and Janssens, G. Pattern-based compaction for ProbLog inference. In *Theory and Practice of Logic Programming, International Conference on Logic Programming, Istanbul, Turkey, 24 - 29 AUGUST 2013* (2013), Cambridge University Press, pp. 1–4.

[75] Shterionov, D., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., and Janssens, G. The most probable explanation for probabilistic logic programs with annotated disjunctions. To appear in *Proceedings of the 24th International Conference on Inductive Logic Programming*.

[76] Skarlatidis, A., Artikis, A., Filipou, J., and Paliouras, G. A probabilistic logic programming event calculus. *TPLP 15*, 2 (2015), 213–245.

[77] Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., and Sato, T. CHR(PRISM)-based probabilistic logic learning. *Theory and Practice of Logic Programming 10*, 4-6 (2010), 433–447.

[78] Stickel, M. A Prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated Reasoning 4* (1987), 353–380.

[79] Sureshkumar, A., Vos, M. D., Brain, M., and Fitch, J. Ape: An AnsProlog* environment. In *In Proceedings of the First International Workshop on Software Engineering for Answer Set Programming, volume 281 of Workshop Proceedings* (2007), pp. 101–115.

[80] Swift, T., and Scott Warren, D. An abstract machine for SLG resolution: Definite programs. In *Proceedings of the 1994 International Symposium of Logic Programming* (November 13-17 1994), pp. 633–652.

[81] Tamaki, H., and Sato, T. Old resolution with tabulation. In *Third International Conference on Logic Programming (ICLP 1986)* (1986), E. Shapiro, Ed., vol. 225 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 84–98.

[82] Taskar, B., Guestrin, C., and Koller, D. Max-margin Markov networks. In *Proceedings of Neural Information Processing Systems* (2003).

[83] Valiant, L. G. Why is Boolean complexity theory difficult? In *London Mathematical Society symposium on Boolean function complexity* (New York, NY, USA, 1992), Cambridge University Press, pp. 84–94.

[84] Van Gelder, A., Ross, K., and Schlipf, J. The well-founded semantics for general logic programs. *Journal of the ACM 38* (1991), 620–650.

[85] Vennekens, J. *Algebraic and logical study of constructive processes in knowledge representation.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, May 2007. Denecker, Marc and De Schreye, Danny (supervisors).

[86] Vennekens, J., Denecker, M., and Bruynooghe, M. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming 9* (2009), 245–308.

[87] Vennekens, J., Verbaeten, S., and Bruynooghe, M. Logic programs with annotated disjunctions. In *Proceedings of International Conference on Logic Programming* (2004), Springer, pp. 431–445.

[88] Vlasselaer, J., and Meert, W. Statistical relational learning for prognostics. In *Belgian-Dutch Conference on Machine Learning* (2012), B. De Baets, B. Manderick, M. Rademaker, and W. Waegeman, Eds., pp. 45–50.

[89] Vlasselaer, J., Renkens, J., Van den Broeck, G., and De Raedt, L. Compiling probabilistic logic programs into sentential decision diagrams. In *Proceedings FLOC Workshop Probabilistic Logic Programming (PLP 2014)* (17 July 2014), p. 12.

Curriculum Vitae

Dimitar Shterionov was born on the 15^{th} of July 1984 in Vratza, Bulgaria. After finishing high school at the math and science High School, "Prof. Emanuil Ivanov", Kjustendil, Bulgaria in 2003 he was accepted to the Bachelor of Informatics program at the Aristotle University of Thessaloniki, Greece with a full scholarship. Before commencing his bachelor studies he first took a one-year mandatory education in the Greek language. He obtained his degree from the Aristotle University of Thessaloniki in 2009. Between 2008 and 2009 he worked as a software developer for Zarzonis Editions, Thessaloniki, Greece and Logodata Pcs., Thessaloniki Greece. In 2009 he was accepted to continue his education at KU Leuven, Belgium where he followed the one-year master program in Artificial Intelligence. During this year he did his master thesis under the supervision of professor Gerda Janssens. In 2010 he joined professor Gerda Janssens in the DTAI (Declaratieve Talen en Artificiele Intelligentie) group at KULeuven for his Ph.D. studies. His main topic during the period of his Ph.D. was the design and development of the probabilistic logic and learning software ProbLog.

List of Publications

Publications in Journals

- Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens and Luc De Raedt, *Inference and learning in probabilistic logic programs using weighted Boolean formulas*, Theory and Practice of Logic Programming (TPLP), 15(3), pages 358-401, 2015.

- Shteliyan Shterionov and Dimitar Shterionov, *Квантитативни методи за изследване на демографското развитие на българските земи през XVII - XIX век*, Publication title in English: *Quantitative methods for investigating the demographic development of the Bulgarian territories in 18^{th} - 19^{th} century.*, Демографската Ситуация и Развитието на България, Edition title in English: Demographic Situation and Development of Bulgaria, (Forum), pages 569-586, 2014, Sofia, Bulgaria, ISBN: 978-954-322-793-8

Publications at International Conferences and Symposia

- Theofrastos Mantadelis, Dimitar Shterionov, and Gerda Janssens, *Compacting Boolean formulae for inference in probabilistic logic programming*, 13^{th} International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2015, 13 pages, 27-30 September, 2015, Lexington, Kentucky, USA

- Dimitar Shterionov, and Gerda Janssens, *Implementation and performance of probabilistic inference pipelines*, 17^{th} International Symposium on Practical Aspects of Declarative Languages, PADL 2015, 15 pages, 18-19 June 2015, Portland, Oregon, USA

- Dimitar Shterionov, and Gerda Janssens, *Crucial components in probabilistic inference pipelines*, 30^{th} Symposium of Applied Computing, SAC 2015, pages 1887-1889, 13-17 April 2015, Salamanca, Spain
- Dimitar Shterionov, Joris Renkens, Jonas Vlasselaer, Angelika Kimmig, Wannes Meert and Gerda Janssens, *The Most Probable Explanation for probabilistic logic programs with annotated disjunctions*, 24^{th} International Conference on Inductive Logic Programming, ILP 2014, 15 pages, 14-16 September 2014, Nancy, France
- Dimitar Shterionov, and Gerda Janssens, *cProbLog: Restricting the possible worlds of probabilistic logic programs*, 1^{st} Workshop on Probabilistic Logic Programming, PLP 2015, pages 12, 17 July 2014, Vienna, Austria
- Dimitar Shterionov, Theofrastos Mantadelis and Gerda Janssens, *Pattern-based compaction for ProbLog inference*, 29^{th} International Conferences of Logic Programming, 4 pages, 24-29 August 2013, Istanbul, Turkey
- Joris Renkens, Dimitar Shterionov, Guy Van den Broeck, Jonas Vlasselaer, Daan Fierens, Wannes Meert, Gerda Janssens and Luc De Raedt, *ProbLog2: From probabilistic programming to statistical relational learning*, Workshop on Probabilistic Programming: Foundations and Applications, in Neural Information Processing Systems, NIPS 2012, 5 pages, 7-8 December 2012, Lake Tahoe, Nevada, USA
- Dimitar Shterionov and Gerda Janssens, *Data acquisition and modeling for learning and reasoning in probabilistic logic environment*, 15th Portuguese Conference on Artificial Intelligence, EPIA 2011, pages 298-312, 10-13 October 2011, Lisbon, Portugal
- Dimitar Shterionov, Angelika Kimmig, Theofrastos Mantadelis and Gerda Janssens, *DNF sampling algorithm for ProbLog inference*, Joint Workshop on Implementation of Constraint Logic Programming Systems and Logic-based Methods in Programming Environments, CICLOPS-WLPE 2010, 15 pages, 14-20 July 2011, Edinburgh, Scotland, UK

Technical Reports

- Dimitar Shterionov, Theofrastos Mantadelis and Gerda Janssens, *Implementation and performance of probabilistic inference pipelines: Data and results*, CW Reports, vol: CW679, 21 pages, March 2015.
- Dimitar Shterionov, Theofrastos Mantadelis and Gerda Janssens, *Pattern-based compaction for ProbLog inference*, CW Reports vol: CW639, 21 pages, July 2013

Printed and published by:
Regalia 6
Sofia, BULGARIA
August, 2015

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
DECLARATIVE LANGUAGES AND ARTIFICIAL INTELLIGENCE (DTAI)
Celestijnenlaan 200A box 2402
B-3001 Heverlee
dimitar.shterionov@cs.kuleuven.be

Regalia 6